How to write EXCITING GAMES FOR THE COMMODORE

Philip O'Carroll

HOW TO WRITE EXCITING GAMES FOR THE COMMODORE 64

Philip O'Carroll

PITMAN

First published 1984

Pitman Publishing Pty Ltd
(Incorporated in Victoria)

158 Bouverie Street
Carlton
Victoria 3053

Level 12
Town Hall House
452-462 Kent Street
Sydney
New South Wales 2000

9th Floor
National Bank Building
420 George Street
Brisbane
Queensland 4000

Associated companies

Pitman Publishing Ltd
London

Copp Clark Pitman
Toronto

Pitman Learning Inc
Belmont, California

Pitman Publishing New Zealand Ltd
Wellington

National Library of Australia

Cataloguing in Publication data

O'Carroll, Philip, 1945– .
How to write exciting games for your Commodore 64.

ISBN 0 85896 181 4.

1. Electronic games. 2. Commodore 64 (Computer) — Programming.
I. Title

794.8'2

Produced for Pitman
by Lucas Comber McGrath Pty Ltd

Printed in Singapore
by GnP Consultants Pte Ltd

Cover design by Lynda Patullo

Contents

Before you start

This is a book for Commodore 64 owners who want to master *sprite* graphics, the wonderful programmable object blocks that are a feature of the Commodore 64 computer.

It shows how to design sprites, how to get them moving, how to detect collisions and how to create explosions and other exciting consequences of their actions.

The fun way to learn about sprites (and all sorts of other things about programming your Commodore 64) is to write an exciting game that demonstrates how everything works.

Some games books give you listings for lot's of programs and explain them in language only the experienced programmer can understand. This book focuses on the development of just one game and explains in detail every step needed for the creation of its graphics, its sound and its action. By the time you've worked your way through it, you'll understand enough about programming in BASIC on the Commodore 64 to be able to write your own games using sprite graphics.

The game itself is called *Commando Sub.* It features seven sprites — moving and stationary, high resolution and multi-color, constant and expanding — and all the sound effects needed really to get the adrenalin flowing. So you can see how everything works and how it all hangs together, it's built up in modules that you can test RUN as you complete them.

There are, of course, a few things you ought to know and a few things you ought to be able to do before you start. You should know how to switch on your computer, connect it to your cassette or disk drive and tune in your monitor. You should know how to SAVE a program and you should know the functions of the various keys on your keyboard. In particular, you should know how to correct typing mistakes using the cursor keys. Beyond that, this really is a book for beginners!

Many of the things that beginners want to know soon after they've started are contained in the *Commodore 64 User's Guide.* If you've worked your way through chapters 1, 2 and 3 of the *User's Guide,* that will be quite enough for you to start on *Commando Sub.* You should, of course, keep your *User's Guide* by your side in case you want to go more deeply into any of the programming techniques we use in *Commando Sub.* It will continue to be a useful reference.

IMPORTANT IMPORTANT IMPORTANT IMPORTANT

Throughout the book you'll find many lines of computer code. When you type them in, your Commodore 64 will automatically start a new line after the 40th character, even if it's in the middle of a word. You should keep typing till you come to the end of the *printed* line before you press the RETURN key. Only press the RETURN key when you want to type a new line number. Aside from that, key every line *exactly* as it's shown. Even a comma left out could cause your program to crash. If you misspell a command or omit a key symbol, your computer will tell you SYNTAX ERROR when you try to RUN the program. But don't despair. It happens to experienced programmers as well as beginners. Just go back to the offending line, correct it and press the RETURN key at the end of the line. Then all will be well.

Finally, if you're not familiar with the symbols that occur from time to time inside PRINT statements, turn to Appendix E. It shows what the symbols mean and how you get them.

1 Introduction and program plan

Commando Sub puts you at the helm of a mini submarine. Your mission is to seek and recover two separate caches of gold and return with them to your home base. The gold, of course, has been stored deep beneath the sea to keep it safe from the reach of the (nasty) imperial forces with whom you are at war and it's needed now to further finance your war effort.

It will be no easy task. A rocky reef and two icebergs prevent your clear sailing and an enemy destroyer lurks above you, randomly dropping deadly depth charges. And as if these hazards were not enough, these waters are home to a fierce sea monster that tends to block the approach to the more valuable cache of gold.

Commando Sub includes a number of subroutines that are easily transportable to a variety of other arcade-style games. The special emphasis, though, is on the handling of sprite graphics. In setting up *Commando Sub* you'll see how to:

- create a variety of sprites;
- set up a background scene using both keyboard characters and sprites;
- program user control of the main sprite;
- program random control of other moving sprites;
- synchronise missile movement with craft movement;
- create realistic sound effects for explosions — a lesser one for stray missiles and a greater one for the exploding sub;
- set up a system that keeps score and remembers old score;
- display game instructions;
- write in joystick control as an alternative to keyboard control.

Further down this page, there's an outline of the structure of the whole program. In describing the step by step writing of the program, however, we don't always follow the same sequence that's given there. We skip around a bit. That's because the order of a finished program isn't always the best order in which to write it. Certainly, it makes it easier if we plan the sorts of modules we're going to need — as we have in the list above — but there are going to be times when we won't know exactly what we need until we're well into our program. Fortunately it's an easy matter then to write the extra lines and then slip them in, back where they were needed.

So read the part called *Structure of the program* to get a bird's eye view of all the modules we're going to need, then hop straight into Chapter 2 — *Designing the submarine.*

Structure of the program — the program plan

Line 47	Instructions
Lines 50-58	Background
Lines 60-65	Start fuel
Lines 70-79	Set up sound
Lines 80-96	Set up sprite variables
Lines 100-194	Set up submarine sprite
Lines 200-490	Set up 'peak' sprite
Lines 500-594	Set up destroyer sprite
Lines 600-694	Set up monster sprite
Lines 700-794	Set up missile sprite
Lines 800-885	Set up explosion sprite
Lines 1000-1050	Submarine control
Lines 1300-1340	Monster movement
Lines 1400-1424	Getting the gold
Lines 2000-2090	Submarine movement routines
Lines 2100-2150	Submarine collision check
Lines 2200-2292	Destroyer movement
Lines 2400-2490	Monster movement routines
Lines 2500-2650	Missile movement
Lines 2700-2780	Missile explode
Lines 2800-2860	Submarine explode
Lines 2900-2940	Enter home base
Lines 3000-3060	Final remarks and score
Lines 3070-3085	Restart and reset certain POKEs
Lines 4000-4085	Instruction subroutine
Lines 6000-6042	Background detail subroutine

2 Designing the submarine

The most important 'character' in our story line is the submarine itself so that will be the first of the sprites that we'll design. In designing any game, though, I find it helps to have the background colors in front of me from the very beginning. So that's where we'll start.

2.1 Background colors

We've got two separate screen areas to deal with: the area underwater and the area above the waterline. One of the these areas will be set as the 'screen color' in the way that we normally think of setting it but the other will have to be an overlay of colored blocks (or reverse spaces). When you display characters over reverse spaces, however, you frequently get complications with the color. The thing to do, then, will be to choose screen color for the area containing the most scenery — that will be the underwater area. We'll use light blue for the sea and cyan for the sky.

The game's action takes place below the surface of the sea so we'll set the waterline fairly high on the screen to leave ourselves plenty of room underwater. The border will be brown so that the brown rocks (to come later) will merge naturally with it.

```
50 REM BACKGROUND COLOR SCHEME
52 POKE53280,9:POKE53281,14:PRINT"[illegible]";
56 FORF=1TO18
57 PRINT"[illegible]                              ";:NEXT
```

Line 50 doesn't actually do anything. It's just a 'header' for this routine. REM (short for REMark) speaks only to the programmer.

Line 52 clears the screen and POKEs the border address with the color code for brown (9) and the screen address with the color code for light blue (14). The reverse heart is the clear screen symbol. To get it you key CLR/HOME together with SHIFT.

Line 56 commences the loop that will print the reverse spaces on the first 9 rows of the screen. Because we've used an 18 cycle loop we only have to include 20 spaces in the PRINT instruction in the next line (a row is 40 spaces long).

Line 57 prints the reverse spaces and colors them cyan. The semi-colon tells the computer to print the spaces one after the other. The word NEXT closes the loop commenced in the previous line. To get the symbols and spaces contained within the quotes, you key:

- RVS ON with CTRL
- CYN with CTRL
- space bar (20 times)

A note about REM statements

You don't need to type in REM statements that appear at the end of a line of computer code. The program works without them. They are there only to help you to understand the instruction itself.

If an instruction consists *solely* of a REM statement you need type only the word REM and not the rest. To omit the line entirely could cause problems since the program might GOTO or GOSUB that line number.

2.2 Creating the sprite

If you look closely, you'll see that every object displayed on your TV screen is made up of a number of little 'bricks'. These are called *pixels*. Each of the regular characters is formed on a space eight pixels wide and eight pixels high. Figure 2.1 shows how the letter 'A' is formed.

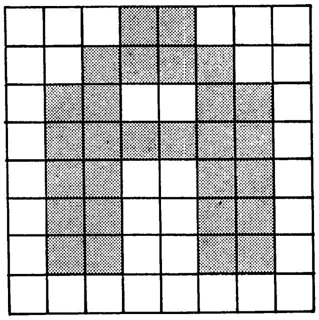


Figure 2.1

This is an 8 × 8 grid. Now for a sprite, we use a 24 × 21 grid like the one shown in Figure 2.2.

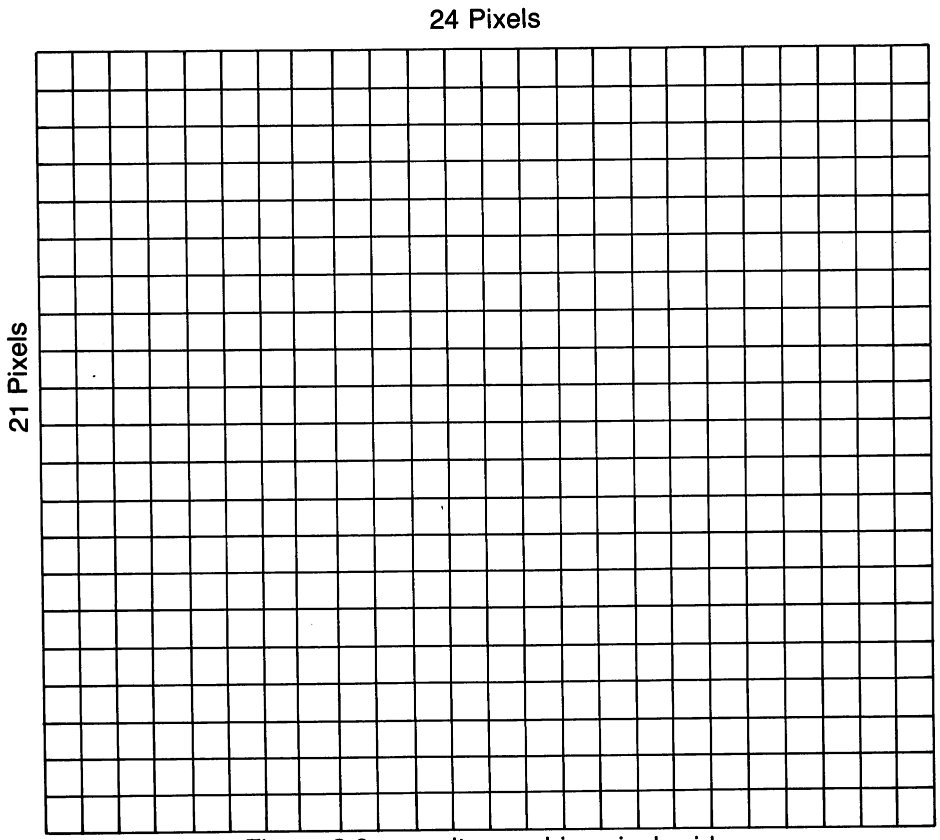


Figure 2.2 — sprite graphics pixel grid

And to create a sprite of our own design, we simply fill in a number of these pixels. Figure 2.3 shows my submarine, laid out on a sprite grid.

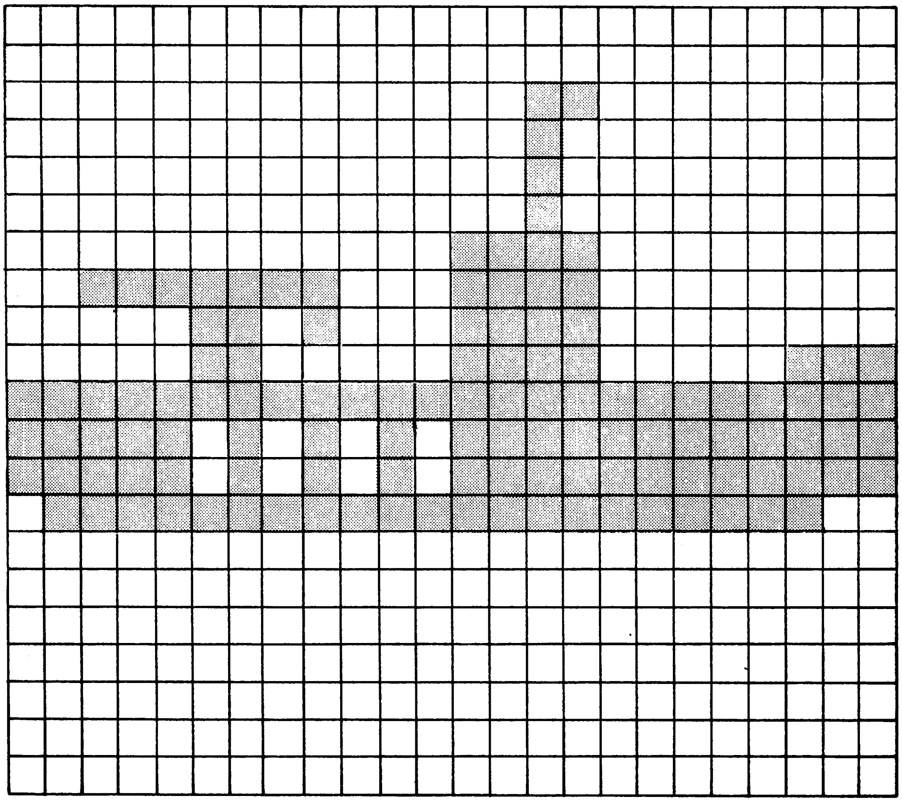


Figure 2.3 — sprite submarine

So far, so good. But how do we tell the computer what we've drawn? How does the computer store and remember the design? To answer those questions we'll have to take a short time out from programming the game itself. The following few pages might look like hard going to start with, but if you read them carefully you will be able to do much more with your machine.

2.3 Making pictures into numbers

Each of the pixels on our TV screen can have only one of two states — it can be on or off. If it's on, we see it as part of a character and if it's off, we don't see it at all. Another way of putting this is to say that when it's on, a pixel has the value of 1 and when it's off, it has the value of 0 (zero). There's a good reason for thinking of it like this: it's the way your computer works. Your computer really only understands in terms of things being on or off — of having a value of 1 or zero. At its heart, it only understands numbers.

In terms of being on or off (1 or 0), then, the pixels in our submarine grid would look the way they are shown in Figure 2.4.

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0
0	0	1	1	1	1	1	1	1	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	0	1	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	0	0	0	0	0	1	1	1	1	0	0	0	0	0	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	0	1	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1
0	1	1	1	1	0	1	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 2.4 — sprite sub in binary bytes

That takes us a little further but we've still got to get all these zeros and ones into our computer's memory. To understand how that's done, we'll need to understand a little of how your computer's memory actually works.

2.3.1 Computer memory

In your computer there are 65536 memory addresses numbered from 0 to 65535 inclusive. Each of these addresses contains eight little flags. Each of these flags can be in either one of only two positions: up or down. You read the flags from left to right. If the leftmost flag is up, the address includes 128; if the rightmost flag is up, that's worth a 1. If both the left and right flags are up and all the rest are down, that memory address holds the number 129 (ie 128 + 1). Here's what each of the flags represents (if it is up):

Flag 7	Flag 6	Flag 5	Flag 4	Flag 3	Flag 2	Flag 1	Flag 0
128 or	64 or	32 or	16 or	8 or	4 or	2 or	1 or
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

(2^7 means $2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2$; 2^6 means $2 \times 2 \times 2 \times 2 \times 2 \times 2$ — and so on)

Remember, though, that you show these numbers by indicating the condition of each of the eight flags at the particular memory address. In other words, you write a row of eight digits, each of which must be either a 0 or a 1. So, for example, if memory address 257 contained 11010001, we would have 209 (128 + 64 + 0 + 16 + 0 + 0 + 0 + 1) in address 257.

2.3.2 Binary numbers and decimal numbers

From the chart above you can see that this whole system of flags is based on the number 2. That's why the string of 0's and 1's is called a *binary* number. Let's convert some binary numbers into ordinary decimal numbers:

10000001 means 128 + 1 = 129

11100000 means 128 + 64 + 32 = 224

00001000 means 8

00000011 means 2 + 1 = 3

We've said that each of the computer's memory addresses contains a set of eight flags which, taken in combination, represent a decimal number. Let's now ask the computer to show us what is actually contained in a particular address. To ask the computer what is contained in any memory address, you type PRINT PEEK and then the address number in brackets. Let's see what's in address 197. Type:

```
PRINT PEEK (197)
```

and hit RETURN. Well, it's a number but it's not a binary number. That's because the makers of your Commodore 64 have thoughtfully arranged to have the computer's binary numbers converted to decimal numbers for you before they're displayed. Most of us recognise decimal numbers a lot more easily than we do binary numbers! To work out which flags are up and down at that address we'd have to break the decimal number down into a binary number. For 64 it would be 01000000 — that is, only flag six would be in the up condition.

More importantly, however, the fact that the computer communicates with us in decimal numbers means that we have to communicate with it in decimal numbers. In other words, we have to convert the zeros and ones which make up our sprite sub into decimal numbers. In order to do that we have to say something about *bits* and *bytes*.

2.3.3 Bits and bytes

Each of the flags we've been talking about is normally called a *bit* and, as we've seen, each address is capable of holding eight bits of information because it has eight flags. The eight bit number is called a *byte*.

Summarising, then, we can say that each address is capable of holding a byte of information where the highest value the byte can have is 255 (all the bits have a 1 in them) and the lowest value is 0 (all the bits contain zero). So every memory address contains a number between 0 and 255, stored as a row of eight bits — each bit a 0 or a 1. (Physically, each byte is a set of eight wiring connections, each of which is either electrically excited or not. When it's excited it has the value of 1 and when it's sleeping, it has the value of 0).

1 BYTE

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
128	64	32	16	8	4	2	1

Total possible value = 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255

Now if we return to our sprite grid full of 0's and 1's, we'll see that it falls neatly into 63 bytes — each byte containing an eight bit binary number. Figure 2.5 shows how the bytes are numbered.

BYTE 0	BYTE 1	BYTE 2
BYTE 3	BYTE 4	BYTE 5
BYTE 6	BYTE 7	BYTE 8
BYTE 9	BYTE 10	BYTE 11
BYTE 12	BYTE 13	BYTE 14
BYTE 15	BYTE 16	BYTE 17
BYTE 18	BYTE 19	BYTE 20
BYTE 21	BYTE 22	BYTE 23
BYTE 24	BYTE 25	BYTE 26
BYTE 27	BYTE 28	BYTE 29
BYTE 30	BYTE 31	BYTE 32
BYTE 33	BYTE 34	BYTE 35
BYTE 36	BYTE 37	BYTE 38
BYTE 39	BYTE 40	BYTE 41
BYTE 42	BYTE 43	BYTE 44
BYTE 45	BYTE 46	BYTE 47
BYTE 48	BYTE 49	BYTE 50
BYTE 51	BYTE 52	BYTE 53
BYTE 54	BYTE 55	BYTE 56
BYTE 57	BYTE 58	BYTE 59
BYTE 60	BYTE 61	BYTE 62

Figure 2.5 — sprite byte grid

It's a reasonably simple matter now to convert each of the eight bit binary numbers on our sprite grid to their decimal equivalents. Figure 2.6 shows what our submarine looks like in decimal numbers.

0	0	0
0	0	0
0	3	0
0	2	0
0	2	0
0	15	0
63	143	0
6	143	0
3	15	7
255	255	255
250	175	255
250	175	255
127	255	252
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0

Figure 2.6 — sprite sub in decimal bytes

Check some of these decimal numbers against the binary originals. Satisfy yourself that the 'translations' are correct. After a while, though, the job of converting binary numbers to decimal numbers can be tedious. You can let your computer do it for you. In Appendix C there's a short program you might find worth keying in when you come to design your own sprites. When you RUN it, it asks you to enter your eight digit binary number and then gives you the decimal equivalent.

2.4 Storing the sprite

Now that we've converted our design to a set of 63 numbers (called DATA) that the computer can understand, our next task is to store those numbers in the computer's memory. Your Commodore 64 has a special place in its memory where it knows to look for sprite DATA. Our first place starts at address 15872 and ends at address 15935 — that is, it consists of a block of 64 memory addresses. The second place then starts at address 15936, the third place at address 16000 and so on. Each place consists of a block of 64 memory addresses.

To tell the computer which block of memory we want to store a sprite in, we have to POKE the sprite's *pointer address* with a pointer number. Each sprite has its own pointer address. The pointer addresses start with 2040 for sprite 0 and end with 2047 for sprite number 7. (There are eight sprites but they're numbered 0 to 7.)

So to tell the computer in which block to store the DATA for sprite 0, we POKE address 2040 with a number which, when multiplied by 64, will equal the first address of the memory block we want to use. If we want to use the block starting at 15872, for example, we'll POKE 2040 with 248 (248 x 64 = 15872). The way we get the 248 is to divide the memory address of the block (15872) by 64.

Now if we POKE 2041 (the sprite pointer address for sprite 1) with the same number (248), sprite 1 will look the same as sprite 0 because the computer will be reading the same set of DATA for it.

So let's put our submarine DATA into a suitable block of memory. The first step will be to locate the memory block for the DATA.

```
100 REM SUB SPRITE
105 SUBMEM=15872:REM 1ST ADDR FOR SUB DATA
107 POKE 2040,248:
```

Line 105 sets up the variable name SUBMEM and gives it the value of the first address in the block of memory in which we want to store our submarine DATA.

Line 107 POKEs the sprite pointer address for sprite 0 with 248, thereby directing the computer to look at address 15872 (248 x 64) to find the DATA for sprite 0.

To get the 63 bytes of DATA into the addresses starting at 15872 we could write a line like this:

```
110 POKE 15872,0:POKE 15873,0:POKE 15874,0
```

and so on for all 63 numbers. But that would be awfully laborious. To make it easier, then, we'll use a FOR. . .NEXT loop and READ the numbers from a DATA list.

```
110 FOR F=0TO62
111 READ BY
112 POKE SUBMEM+F,BY
113 NEXT
120 DATA0,0,0,0,0,0,0,3,0,0,2,0,0,2,0
121 DATA0,15,0,63,143,0,6,143,0,3,15,7
122 DATA255,255,255,250,175,255,250,175
123 DATA255,127,255,252,0,0,0,0,0,0,0,0,0
124 DATA0,0,0,0,0,0,0.0,0,0,0,0,0,0,0
```

Line 110 sets up the loop that will cause the computer to repeat the instructions between the FOR in line 110 and the NEXT in line 113, 63 times.

Line 111 instructs the computer to take the next (previously unread) item of DATA from lines 120 to 124 and to store it as a variable called BY.

Line 112 POKEs the variable SUBMEM (15782) plus F (the number of the cycle on which the loop is) with BY (the item of DATA just READ). In other words, every time the POKE instruction is carried out, 1 is added to the previous address so the next item of DATA will go into an address one more than the last one. In this way, each of the 63 items of DATA in lines 120 to 124 will be placed in the 63 addresses commencing at SUBMEM (15872).

Lines 120 to 124 contain the decimal numbers from our sprite grid. Notice that each is separated from the previous one by a comma and that they are in the same order as the bytes of our sprite grid. Count them to make sure you've got all 63!

Now, of course, you want to see the sub on the screen! Before it can be displayed, however, there are several vital statistics your C-64 needs to know about it. In Chapter 3 we'll deal with those statistics and display the sub that's now safely stored in your computer's memory.

3
Enter the submarine

3.1 Essential facts about our sprite

In addition to the DATA that defines the shape of our sprite, we have to tell our computer a number of other things about it before it can be displayed on the screen. These things are:

- its color
- its priority (ie — whether it's to appear in front of or behind other objects on the screen)
- its initial location.

3.2 Color

We store the color codes for the eight sprites at addresses 53287 to 53294. Fortunately, however, we don't have to remember the particular address for each sprite. If we can remember 53287, that will be enough because we can then say that the address for the color code of sprite 0 is 53287 + 0 (53287), the address for the color code of sprite 1 is 53287 + 1 (53288) — and so on for each of the sprites. In other words, we just have to add the sprite number to 53287 to get the address for the color code of the sprite in question. And that, of course, means we can set up an abbreviation for the color code addresses of all the sprites we might want to use in our program:

```
80 REM SPRITE DEFINITIONS
90 SC=53287:REM SPRITE COLOR
```

Line 90 sets up a variable name (SC) for the address for sprite color. From now on, to color any sprite, we'll only have to POKE SC (plus the number of the sprite in question) with the code of the color we want.

Now we can assign sprite 0 its color. Let's say dark grey:

```
180 REM ASSIGN A COLOR
181 POKE SC+0,11:REM SPRITE COLOR 0
```

Line 181 POKEs the variable SC + 0 (the color address for sprite 0) with 11 — the color code for dark grey.

3.3 Background priority

Our next step is to decide whether we want our sprite to appear *in front of* or *behind* the background. We store this information for each sprite at address 53275. To make life easier when we come to our other sprites later in the program, then, we'll start by abbreviating the address:

```
91 BP=53275:REM BACKGROUND PRIORITY
```

Line 91 sets up the variable name BP for the address for a sprite's priority over the background.

Now we can assign sprite 0 its priority:

```
182 REM ASSIGN BACKGROUND PRIORITY
183 POKE BP,0:REM SUB IN FRONT OF BACKGROUND
```

Line 183 POKEs the variable BP (the address for sprite priority over the background) with 0. This, in fact, gives *all* eight sprites priority over the background. Later on in the program we'll want to make some sprites appear to move behind parts of the background scenery — but we'll deal with that when we come to it.

3.4 Initial location

Now we must decide where we'd like our sub to begin when the game starts. Let's say on the surface, at the left of the screen. To tell the computer this location, we have to know how the rows and columns of pixels are numbered.

3.4.1 Measuring in pixels

As we saw earlier with our diagram of the letter A, each keyboard character takes up 8 rows by 8 columns of pixels. Now we also know that our screen display area consists of 40 characters across by 25 characters down. Measured in pixels, then, our screen is 320 pixels across (40 x 8) and 200 pixels down (25 x 8). Notice that we're talking about the screen area within the borders.

Where sprites are concerned, though, the *full screen* is available for movement — ie the regular display area *and* the border. That means our sprite can go off screen and still be there hiding! So the screen dimension on which we must initially locate our sprite is actually a pixel grid 368 by 255.

The sprite itself is always located by its upper left hand pixel — even if this pixel is not 'colored in'.

3.4.2 Storing X and Y values

To locate anything on our screen grid, we have to quote two numbers:

- the column number of the pixel (how many pixels across the screen) and
- the row number of the pixel (how many pixels down the screen).

With those two numbers, we can pinpoint exactly the location for the upper left pixel of our sprite. For convenience, we call the column positions the X axis of our sprite and the row positions, the Y axis.

One memory address can easily hold the vertical position (the Y axis) of a sprite. But alas, two POKEs may be needed to tell the computer where you want your sprite along its horizontal (X) axis. This is because there are 344 X positions to choose from. (Notice we've said 344 X positions, not 368. This is because the sprite itself is 24 pixels wide so its furthest position to the right can only be 368 –24 pixels).

We need two POKEs for the X axis because the maximum number any single address can hold is 255. We provide for the two POKEs by dividing the screen into two areas: a left area covering from column 0 to column 255 and a right area, *renumbered* 0 to 91. To tell the computer which side of the screen we're dealing with, we POKE address 53264 with either a 0 or a 1: a 0 for the left hand side and a 1 for the right hand side. As a first step, let's abbreviate our side of screen address:

```
93 SS=53264:REM SIDE OF SCREEN
```

and then POKE SS with a zero because we're starting our sprite over on the left hand side of the screen.

```
184 REM ASSIGN INITIAL POSITION
185 POKE SS,0:REM LEFT OF SCREEN
```

Now, I would like to start our sub at column 31, row 110. How do we tell the computer this? You guessed it: there is a particular address for the X position and the Y position for each sprite. To make life simple for ourselves, we'll call the X addresses X(0) to X(7) (where the numbers in the brackets are the sprite numbers) and the Y addresses Y(0) to Y(7). The X addresses start at 53248 and go up in two's. The Y addresses start at 53249 and also go up in two's. Here's how we assign the actual addresses to our X and Y variables:

```
94 FORF=0TO7:X(F)=53248+2*F:NEXT:REM X ADDRS
96 FORF=0TO7:Y(F)=53249+2*F:NEXT:REM Y ADDRS
```

Line 94 creates a FOR. . .NEXT loop and then sets up the variable X *subscripted* by each of the cycles in the loop. X(F) is like saying X_0, and so on up to X_7. X(F) is then given the value of the starting address for values of X *plus* two times F (the number of the cycle on which the loop is). In this way each of the subscripted variables is given a value two more than the previous one. Line 96, of course, does the same thing for the Y variable.

Using our subscripted variables for sprite 0 we can now tell the computer where to put the sub:

```
186 POKE X(0),31:REM 31ST COLUMN
188 POKE Y(0),110:REM 110TH ROW
```

3.5 Enabling the sprite

After all this, you'd think the sub would appear. But no, there is one more command needed. It's a command that enables us to display (or not display) any object we've created. It's like an ON/OFF switch that doesn't affect the actual storage of all the details of design, color and so on that we've previously entered. The 'switch' for sprite 0 is a POKE to address 53269. For ON we POKE it with the number 1 and for OFF we POKE it with 0. As usual, our first step will be to set up an abbreviation for the address. We'll call the new variable EN (for ENable).

```
92 EN=53269:REM SPRITE ENABLE - ON/OFF
```

And to use it for our sub:

```
190 POKE EN,1:REM SET SPRITE 0 ON
```

Warning: before you RUN a program, be sure to SAVE it. One wrong number could make it crash irretrievably (all that typing all over again . . . groan).

Now you can RUN the program so far. At last — enter the sub!

4
Animating the submarine

4.1 Keyboard control

At the moment our submarine is sitting motionless in its place at the top left of the screen. We have to get it moving. Our first step will be to choose control keys for its movement left, right, up and down. We'll use key A for left, key D for right, and the function keys f5 for up and f7 for down. We find the code numbers for each of these keys from the chart shown as Figure 4.1.

#	key	#	key	#	key	#	key
0	DEL	16	5	32	9	48	£
1	RETURN	17	R	33	I	49	*
2	⇆	18	D	34	J	50	;
3	f7	19	6	35	Ø	51	HOME
4	f1	20	C	36	M	52	none
5	f3	21	F	37	K	53	=
6	f5	22	T	38	O	54	↑
7	↑↓	23	X	39	N	55	/
8	3	24	7	40	+	56	1
9	W	25	Y	41	P	57	←
10	A	26	G	42	L	58	none
11	4	27	8	43	−	59	2
12	Z	28	B	44	.	60	none
13	S	29	H	45	:	61	none
14	E	30	U	46	@	62	Q
15	none	31	V	47	,	63	none

Figure 4.1 — keyboard PEEKs held at 197

A is key number 10, D is key number 18, f5 is key number 6 and f7 is key number 3. If you want to, of course, you can choose to use a different set of keys for your controls. Just convert them to their corresponding code numbers.

Now we must instruct our computer what to do when it recognises any of these keys being pressed. We'll have to test for these keys and use a subroutine for each direction.

```
1000 REM CONTROLLING SUB
1010 KE=PEEK(197):REM SCAN KEYBOARD
1015 IF KE=6 THEN GOSUB2020:REM F5 UP
1020 IF KE=3 THEN GOSUB2040:REM F7 DOWN
1025 IF KE=10 THEN GOSUB2060:REM A WEST
1030 IF KE=18 THEN GOSUB2080:REM D EAST
1300 GOTO1010
```

Line 1010 sets up the variable name KE for the contents of memory location 197 (the place that holds the code number of the key being pressed).

Lines 1015 to 1030 send the computer to the subroutines that will cause the sub to move, according to which key has been pressed. We haven't written those subroutines yet but we will do shortly.

Line 1300 sends the computer back to the start of the keyboard scan routine to check for the next key press.

4.2 Vertical movement

Now for the movement subroutines. First of all we'll deal with the sub's vertical movement — ie up and down:

```
2000 REM SUB MOVEMENT (= SPRITE NUMBER 0)
2020 REM SPRITE UP
2022 PY=PEEK(Y(0)):REM PRESENT Y POSN
2025 POKE Y(0),PY-4:RETURN:REM RAISE Y POSN
2040 REM SPRITE DOWN
2042 PY=PEEK(Y(0)):REM PRESENT Y POSN
2045 POKE Y(0),PY+4:RETURN:REM LOWER
```

Line 2022 sets up the variable name PY for sprite 0's present Y position stored at Y(0).

Line 2025 POKEs the sprite's vertical position stored at Y(0), with its present position minus 4 — ie it raises it 4 rows of pixels up the screen. The command RETURN then takes the computer back to line 1300 to check for the next key press.

Lines 2042 and 2045 do the same thing with just one difference. Line 2045 adds 4 rows to the sprite's present position to make it go down.

Now you can test RUN the program so far and test keys f5 and f7 for their control of vertical movement. Warning: if your Y value does not stay within 0 and 255 (ie — if you sail above or below the screen display area) you'll get the message ILLEGAL QUANTITY ERROR. A byte cannot contain values ouside these limits.

The speed of the sub's movement is determined by the –4 and + 4 in lines 2025 and 2045. You can, if you want to, speed up or slow down the sub's vertical movement by increasing or decreasing these numbers.

4.3 Horizontal movement west

The subroutines for horizontal (east-west) movement are dealt with in much the same way as the subroutines for vertical movement. To move west (left) we'd assume we could simply write:

```
2060 REM SPRITE WEST
2062 PX=PEEK(X(0)):REM PRESENT X POSN
2070 POKE X(0),PX-5:RETURN:REM GO WEST
```

where line 2070 moves the X position 5 pixels to the left for a press of key A. Now that's all right as far as it goes but there is a complication to be dealt with. When we pass from the left area of the screen to the right area, we have to change the side of screen (SS) number. Recall that the left area contains columns 0 to 255 and the right area, a set of columns renumbered from 0 to 91. In effect, for the right area, this means we've reset the X value to zero.

The byte stored at SS (our side of screen address) has, like all bytes, 8 bits. Each of these bits holds a 0 or a 1 that says which side of the screen each of sprites 0 to 7 occupies. Bit 0 corresponds with sprite 0, bit 1 corresponds with sprite 1, and so on. A zero in the appropriate bit number places the sprite in the left area of the screen and a 1 places it in the right area.

So let's assume our sprite is in the right hand part of the screen and we want to move it to the left. We start by asking the computer to check which side sprite 0 is on:

```
2064 SP=PEEK(SS):PS=SPAND1:REM SIDE OF SCR
```

Line 2064 lets the variable SP equal the contents of SS (the side of screen byte). It then sets up another variable, PS (for present side of screen) and gives it the value of SP AND 1. This is the value of bit 0 of the byte because any byte AND 1 is equal to the value of bit 0.

Any byte AND 1 = the value of bit 0.

So if bit 0 of SS is 1 then PS = 1 — which means our sub is right of screen. And if bit 0 of SS is 0 then PS = 0 — which means that our sub is left of screen. Now we'll never let the sub go to the extreme west (column 0). So if PX = 0 when we're heading west we must be changing from east to west side of the screen.

```
2068 IF PX=0 THEN 2073
2069 REM LINE 2068 IS FOR CROSSING TO WEST
```

where line 2068 sends the computer on to line 2073 (the instructions for moving to the west) IF the sub's present position is in the eastern part of the screen — ie columns 0 to 91.

```
2073 POKE X(0),251:POKE SS,PEEK(SS)AND254
```

Line 2073 is the interesting part. First of all, if our X value has reached 0 while we've been moving west, it must be that we've crossed the imaginary line between east and west. We jump 5 columns at a time in east-west motion so the first position on re-entry to the west is column 251. So we POKE sprite 0's X position into 251 — POKE X(0),251.

Now we have to reset our side of screen (SS) to zero for sprite 0. That means we have to POKE a 0 into bit 0 of SS. If we just say POKE SS,0, however, we put a 0 into *all* the bits of SS and that puts *all* the sprites over on the left. We don't want to do that when we only want to move the sub so the challenge will be to POKE a 0 at bit 0 of SS without upsetting whatever 0's and 1's are already at bits 7 to 1.

We do it like this. Firstly, we PEEK(SS) to give us the byte that is already in SS. Then to change just bit 0, we say AND 254. AND 254 is equal to all bits except bit 0. In binary presentation it looks like this: 11111110. The second half of line 2073 then becomes POKE SS, PEEK(SS)AND 245.

Any byte AND 254 = all bits except bit 0

It's worth pausing for a moment to see how the same thing is done for sprites other than sprite 0. We'll be using those other sprites later so it's as well to include the explanation now. If we were crossing to the west side for, say, sprite 3 we'd have to alter bit 3 of SS to 0. To do that we'd say PEEK(SS)AND 247. The reasoning becomes clear when we see it in binary: If PEEK(SS) was already 10101010, we'd have:

```
PEEK(SS)   10101010
AND 247    11110111
           --------
TOTAL      10100010
           --------
```

where the only digit in the TOTAL that's different from the value in PEEK(SS) is the digit fourth from the right — ie bit 3.

4.4 Horizontal movement east

Eastward movement follows the same pattern. At least, as we see, down to line 2088. The only other difference is that we're adding 5 places instead of taking them away.

```
2080 REM SPRITE EAST
2082 PX=PEEK(X(0)):REM PRESENT X POSN
2084 SP=PEEK(SS):PS=SPAND1:REM SIDE OF SCR
2088 IF PX=251 THEN 2076:REM CROSS TO EAST
2090 POKE X(0),PX+5:RETURN:REM GO EAST
```

Line 2088 is similar to line 2068 except that our sprite is assumed to be in column 251 — the last position in the western sector it may occupy before it 'jumps' into the eastern sector. Line 2088 sends the computer to line 2076 (the instructions for moving east) IF the sub's present position is column 251.

```
2076 POKE X(0),0:POKESS,PEEK(SS)OR1:GOTO2075
2075 RETURN
```

Line 2076 is the interesting part of this sequence. First of all, if our X value has reached 251 while we've been moving east, it must be that any further movement will take it into the eastern sector and for that, we have to reset the X value to 0. So we POKE X(0),0.

Line 2075 will serve as the exit line back to sub control.

Now we have to reset our side of screen (SS) to 1 for sprite 0. That means we have to POKE a 1 into bit 0 of SS, without changing any of the other bits in the byte. We do it like this. Firstly, we PEEK(SS) to give us the byte that is already in SS. Then to change just bit 0, we say OR 1. OR 1 affects only bit 0. It becomes 1. In binary presentation it looks like this: 000000001. The second half of line 2076 then becomes POKE SS, PEEK (SS)OR 1.

Any byte OR 1 will put a 1 at bit 0.

Again, the reasoning becomes clear when we see it in binary. Let's say the byte in SS was already 11001100. We'd have:

PEEK(SS)	11001100
OR 1	00000001
TOTAL	11001101

where the only digit in the TOTAL that's different from the value in PEEK(SS) is the digit in bit 0 — the one at the far right.

Now we can SAVE and test RUN for movement east and west. Don't expect your sub to move quite as quickly as this in the game itself: many more steps will occur between every sub position and that will slow it down.

4.5 Limits to movement

You should now be able to drive the sub up and down, left and right. Accidental movement outside of the screen limits, however, will cause you frustration because your program will stop and give you an ILLEGAL QUANTITY error message every time. Our next step, then, will be to build in some limits to movement. And, of course, we'd better prevent our sub from being able to fly through the sky!

Our first step will be to make a variable name for these limits to sprite movement. It will make our work easier later on. We'll use U(0) to U(7) for the upper limit of movement for the sprites 0 to 7 and L(0) to L(7) for the lower limit. For the limit to western movement we'll use W(0) to W(7) and for eastern movement, E(0) to E(7). The limits to keep our sub safely in the water will be:

```
192 U(0)=110:L(0)=230:REM UP/LOW LIMITS SUB
194 W(0)=31:E(0)=60:REM SUB EAST/WEST LIMITS
```

Now we must include them in our movement subroutines:

```
2024 IF PY=U(0) THEN RETURN:REM UPPER LIMIT
2044 IF PY=L(0) THEN RETURN:REM LOWER LIMIT
```

Lines 2024 and 2044 will prevent movement above the waterline and below the sea floor. Notice that for this vertical movement we're dealing with the Y value of the sprite.

For the east-west limits we'll need:

```
2066 IF PX<=W(0)ANDPS=0 THENRETURN:REM W LIMIT
2086 IF PX=E(0)ANDPS THENRETURN:REM EAST LMT
```

Notice that we've again got to keep our side of the screen in mind when we're dealing with east-west movement. In line 2066, the AND PS=0 means that we're in the western sector. In line 2086, the AND PS means we're in the eastern sector.

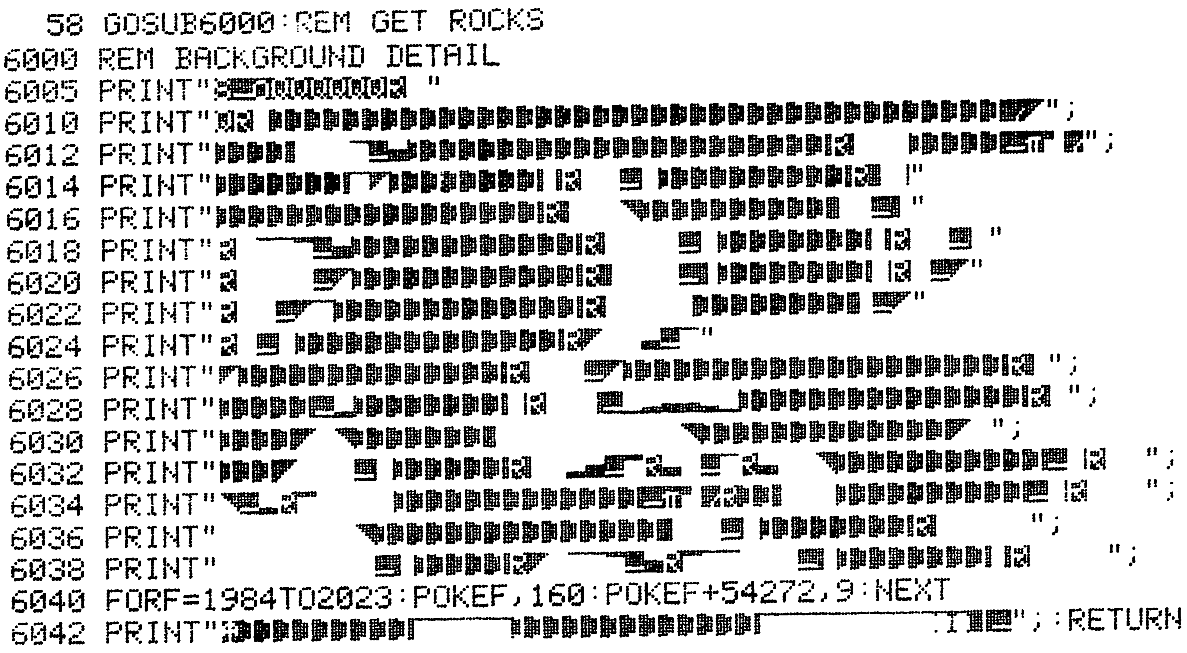
SAVE and test RUN again. Drive anywhere: you can't get into trouble!

5
The undersea stage

Well, there's not much challenge in drifting through clear waters so let's add the props that will bring this part of the ocean deep to life. We'll have a rocky reef jutting from the sea floor, massive icebergs hanging their bulk into the upper levels, a home base and, of course, the gold itself.

5.1 Rocks, gold and home base

These props are stationary and look good enough constructed of the regular keyboard characters. The easiest way of creating them is with the simple command PRINT. To make it easier for you to read and retype the lines of code, the printout is followed by a line by line explanation of the key presses you need. Don't type the colons that separate the key presses!



```
  58 GOSUB6000:REM GET ROCKS
6000 REM BACKGROUND DETAIL
6005 PRINT"[illegible] "
6010 PRINT"[illegible]";
6012 PRINT"[illegible]";
6014 PRINT"[illegible] !"
6016 PRINT"[illegible]"
6018 PRINT"[illegible]"
6020 PRINT"[illegible]"
6022 PRINT"[illegible]"
6024 PRINT"[illegible]"
6026 PRINT"[illegible] ";
6028 PRINT"[illegible] ";
6030 PRINT"[illegible] ";
6032 PRINT"[illegible] ";
6034 PRINT"[illegible] ";
6036 PRINT"          [illegible] ";
6038 PRINT"           [illegible] ";
6040 FORF=1984TO2023:POKEF,160:POKEF+54272,9:NEXT
6042 PRINT"[illegible]";:RETURN
```

```
6005  PRINT:QUOTE:HOME:RVS OFF:DARK GREY:CRSR ↓(6 times): RVS
      ON:SPACE:QUOTE

6010  PRINT:QUOTE:CRSR ↓:RVS ON:COMMODORE N:CRSR →
      38 times):BROWN:SHIFT£:QUOTE:SEMI-COLON

6012  PRINT:QUOTE:CRSR → (4 times):WHITE COMMODORE
      H:SPACE:SPACE:COMMODORE Y:RVS OFF:COMMODORE I:BROWN:
      CRSR → (22 times)WHITE:RVS ON: COMMODORE H:SPACE:SPACE:
      BROWN CRSR → (5 times):RVS OFF: YELLOW:COMMODORE V:
      BROWN:QUOTE:SEMI-COLON

6014  PRINT:QUOTE:CRSR → (7 times):WHITE:COMMODORE Y:SHIFT
      £:BROWN:CRSR → (8 times):COMMODORE L:RVS ON:SPACE:SPACE:
      RVS OFF:COMMODORE H:CRSR→ (10 times):WHITE:RVS ON:
      COMMODORE K:COMMODORE M:BROWN:QUOTE

6016  PRINT:QUOTE:CRSR→ (17 times):RVS ON:COMMODORE J:
      SPACE:SPACE:COMMODORE*:CRSR→ (10 times):WHITE:
      COMMODORE J:SPACE:RVS OFF:COMMODORE K:BROWN:QUOTE

6018  PRINT:QUOTE:RVS ON :SPACE:COMMODORE T,Y,U:RVS OFF:
      COMMODORE I:CRSR → (12 times):RVS ON:COMMODORE G:SPACE
      (3 times):RVS OFF:COMMODORE G:CRSR → (8 times):WHITE:
      COMMODORE L:RVS ON: SPACE (twice):RVS OFF:COMMODORE G:
      BROWN:QUOTE

6020  PRINT:QUOTE:RVS ON: SPACE (4 times):RVS OFF:SHIFT £:CRSR →
      (12 times):RVS ON:COMMODORE K: SPACE (3 times):RVS OFF:
      COMMODORE K:CRSR → (8 times):WHITE:COMMODORE N:RVS ON:
      SPACE:RVS OFF:SHIFT £:BROWN:QUOTE

6022  PRINT:QUOTE:RVSON: SPACE (twice):RVS OFF:SHIFT £:
      COMMODORE Y:CRSR → (13 times):RVS ON:COMMODORE G:SPACE:
      (3 times):COMMODORE L:CRSR → (8 times):WHITE:COMMODORE J:
      RVS OFF:SHIFT £:BROWN:QUOTE

6024  PRINT:QUOTE:RVS ON:SPACE:RVS OFF:COMMODORE H:CRSR →
      (14 times):RVS ON:SHIFT £:2 SPACES:COMMODORE O:RVS OFF:
      COMMODORE Y:QUOTE

6026  PRINT:QUOTE:SHIFT £:(twice) CRSR → (14 times):RVS ON:
      COMMODORE H: SPACE (twice):RVS OFF:SHIFT £:CRSR → (20 times):
      RVS ON:COMMODORE J:QUOTE:SEMI-COLON

6028  PRINT:QUOTE:CRSR → (5 times):RVS OFF:COMMODORE P: CRSR →
      (8 times):COMMODORE N:RVS ON:RVS OFF: SPACE: (twice):
      COMMODORE L:COMMODORE @,P,O,O,P,@:CRSR → (15 times):
      RVS ON:COMMODORE H:QUOTE:SEMI-COLON
```

6030 PRINT:QUOTE: → CRSR (4 times):SHIFT £:SPACE:COMMODORE*: CRSR → (7 times):COMMODORE J: SPACE:(9 times):COMMODORE*: CRSR → (13 times):SHIFT £:SPACE:QUOTE:SEMI-COLON

6032 PRINT:QUOTE:CRSR → (3 times):SHIFT £: SPACE (3 times):RVS OFF: COMMODORE G:CRSR → (6 times):RVS ON:COMMODORE G:SPACE: COMMODORE P,I:RVS OFF:COMMODORE U:RVS ON:COMMODORE O:
SPACE:RVS OFF:COMMODORE Y:RVS ON:COMMODORE O:CRSR → (11 times):RVS OFF:COMMODORE N: RVS ON:SPACE (twice):QUOTE: SEMI-COLON

6034 PRINT:QUOTE:COMMODORE *:RVS OFF:COMMODORE P:RVS ON: COMMODORE U: SPACE (4 times):CRSR → (13 times):RVS OFF: YELLOW:COMMODORE V:BROWN:RVS ON:CRSR → (twice): COMMODORE H: SPACE (twice):CRSR → (10 times):RVS OFF: COMMODORE N: RVS ON: (7 times) SPACE (3 times):QUOTE: SEMI-COLON

6036 PRINT:QUOTE: SPACE (7 times):COMMODORE *:CRSR → (15 times): COMMODORE K: SPACE (twice):RVS OFF:COMMODORE H:CRSR → (8 times):RVS ON:COMMODORE H: SPACE (4 times):QUOTE: SEMI-COLON

6038 PRINT:QUOTE: SPACE (8 times):RVS OFF:COMMODORE J:CRSR → (5 times):RVS ON:SHIFT £:SPACE:COMMODORE T,Y,U:RVS OFF: COMMODORE O:RVS ON:COMMODORE U,Y,T: SPACE (3 times): RVS OFF:COMMODORE J:CRSR → (8 times):COMMODORE L:RVS ON: SPACE (4 times):QUOTE:SEMI-COLON

6040 TYPE AS SHOWN

6042 PRINT:QUOTE:MID GREY:CRSR → (9 times):COMMODORE T (5 times): CRSR → (13 times):COMMODORE T (9 times):CRSR ↑ (twice):CRSR ← : RVS OFF:QUOTE:SEMI-COLON:COLON the word RETURN

Line 6040 is the only one for which we've used POKEs rather than PRINT. That's because, when we finish PRINTing the bottom line of any piece of wörk, the computer automatically scrolls it up. Were that to happen in this case, we'd have our rocks suspended over the sea floor!

In line 6042 we've then gone over the POKEs of line 6040 with two mud patches. It's safe to PRINT these because we don't go to the end of the line with them. We finish with an up cursor and a semi-colon and this makes the computer think we are not going any lower.

Now you can SAVE and test RUN.

5.2 Getting the gold

If you look closely at the underseascape you now have displayed, you'll see two small yellow blocks in an otherwise mostly brown design. These are the caches of gold. They were created in lines 6012 and 6034.

To retrieve the gold, the captain of the sub must bring the front deck of his craft under it. With that action it's automatically loaded. So our first step will be to include a scan for the positions holding the gold within our sub control routine. If the gold is 'loaded' we must:

- Record the fact that we got the gold. If it was the top gold then TG = 1. If it was the bottom gold, then BG = 1.
- Print our success at the top of the screen.
- Erase the gold from where it was collected.

```
1046 IF PY=126 AND PX=60 THEN GOSUB 1410
1048 IF PY=214 AND PX=166 THEN GOSUB1420
```

Lines 1046 and 1048 contain the Y and X coordinates for the caches of gold: 126 and 60 for the gold in the east and 214 and 166 for the gold in the deep. They direct the program to the gold-getting routines IF the sub's position is defined by the coordinates.

```
1400 REM GETTING THE GOLD
1410 IF(SP AND 1)=0 THEN RETURN
1411 REM LINE 1410 CHECKS SIDE OF SCREEN
1412 POKE 1463,32:REM SPACE OVER GOLD
1414 PRINT"▒",,,"▗▛▌EAST GOLD";:TG=1:RETURN
1420 IF(SP AND 1) THEN RETURN
1421 REM LINE 1420 CHECKS SIDE OF SCREEN
1422 POKE1884,32:REM SPACE OVER GOLD
1424 PRINT"▒",,"▗▛▌▌▙DEEP GOLD":BG=1:RETURN
```

5.3 Entering home base

The only other non-brown characters are the two grey blocks at the surface left of the screen. These represent the sub's home base. They were created in line 6005. We want the sub to be able to enter this base — that is, to go off screen. This means we'll have to cancel our limit to movement west when we reach the waterline. Here is a checking routine to see if the sub has reached surface — that is, its upper limit U(0).

```
2066 IF PX<=W(0)ANDPS=0 THEN2910:REM W LIMIT
2900 REM WEST LIMIT FOR SUB
2920 PY=PEEK(Y(0)):REM PRESENT Y POSN
2924 IF PY<>U(0) THEN RETURN:REM UPPER LIMIT
```

Notice that we've edited line 2066 so that it now sends the computer to this checking routine rather than straight back to the lines that check for the next press of a control key. Line 2924 RETURNs to that sequence IF the sub's vertical position (PY) is anything other than the upper limit, U(0).

If we have reached the surface, though, the computer will move straight on to the next line. In that line we'll have to change the west limit to a position off screen. The last screen position for the sub occurs when PX = 6 so we have to add one more position to it:

```
2930 IF PX=6 THEN POKE X(0),1:GOTO 3000
3000 GOTO 3000
```

Line 3000 is just a 'tail chaser' so we can test run our program. At a later stage we'll use it as the starting line of our finishing sequence.

Now if the sub is already part way off screen, we want it to continue its progress so it's all the way off. It's our regular line for westward movement and RETURNs the computer to the control sequence to check for the next key press when it's completed.

```
2940 POKE X(0),PX-5:RETURN:REM GO WEST
```

Note that at this point we cannot allow vertical movement. To prevent it we'll add:

```
2910 IF PX<W(0)THEN POKE Y(0),110:REM UP LMT
```

where we tell the computer to POKE the sub's Y position with the upper limit (110) IF its X position is less (ie — further west) than the regular western limit — W(0).

Now go for a test RUN. SAVE your program first in case there's a typing error in it. You should be able to enter home base.

5.4 The tips of the icebergs

For the present it's safe for our sub to collide with the land masses and the tails of the icebergs because we've not yet written the sequences for collision detection. They will come later.

Even then, however, what we've got now won't prove to be much of a hazard because as you have probably noticed, the sub can float right over the tops of them. To prevent this, we'll add tips to the icebergs and a peak to the central reef. With keyboard characters we're limited to fairly geometric sorts of shapes so we'll do it with a sprite that we can sit on top of each of the icebergs and the central reef. The tip (or peak) is just an irregular shape 21 pixels high and 24 pixels wide, with a flat bottom. If you like, you can design your own and fit it into this program.

The design for my peak is shown in Figures 5.1, 5.2 and 5.3.

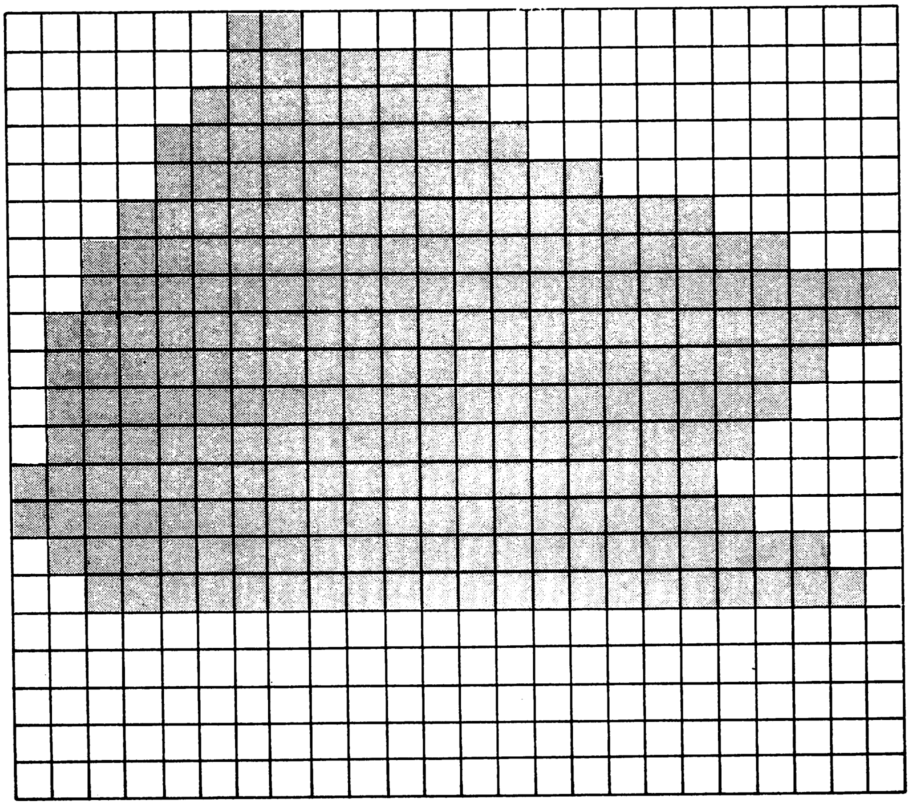


Figure 5.1 — sprite 'peak'

0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0
0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0
0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0
0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 5.2 — sprite ‘peak’ in binary bytes

3	0	0
3	240	0
7	248	0
15	252	0
15	255	0
31	255	224
63	255	248
63	255	255
127	255	255
127	255	252
127	255	248
127	255	240
255	255	224
255	255	240
127	255	252
63	255	254
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0

Figure 5.3 — sprite ‘peak’ in decimal bytes

And here it is as part of the program:

```
200 REM LEFT ICEBERG
205 R1MEM=15936
207 POKE 2041,249:REM SPRITE POINTER #1
208 REM DIRECTED TO 249*64, THAT IS 15936
210 FOR F=0TO62
211 READ BY
212 POKE R1MEM+F,BY
213 NEXT
220 DATA3,0,0,3,240,0,7,248,0,15,252,0
221 DATA15,255,0,31,255,224,63,255,248
222 DATA63,255,255,127,255,255,127,255,252
223 DATA127,255,248,127,255,240
224 DATA255,255,224,255,255
225 DATA240,127,255,252,63,255,254
226 DATA0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
280 REM ASSIGN A COLOR
281 POKE SC+1,1:REM SPRITE COLOR 1
282 REM ICEBERG APPEARS IN FRONT OF SKY
283 REM BP STILL HAS ALL ZEROS FROM LINE 183
284 REM ASSIGN INITIAL POSITION
285 POKE SS,0:REM LEFT OF SCREEN
286 POKE X(1),56:REM 56TH COLUMN
288 POKE Y(1),114:REM 114TH ROW
290 POKE EN,3:REM SET SPRITE 0 & 1 ON
```

SAVE your program and give it a test RUN.

Apart from the numbers in the DATA lines, the routine for the sprite is virtually the same one we used for our sub. Our job was simpler this time, however, because the iceberg does not have to move. These points are worth noting:

- We POKEd 3 into our enabling byte EN. In binary, this gives us 00000011 and enables both sprite 1 and sprite 0 (because both bit 1 and bit 0 are now 1).
- We've given the address for sprite 1's DATA the variable name R1.

Now for the iceberg on the right, all we have to do is set up sprite 2 and point it to the same address (15936) for its DATA. That means we don't have to retype the DATA lines. We just have to remember to POKE a 1 into the third bit of the ENabling byte (00000111 = 7) to set it on — and, of course, to locate it in the right place.

```
300 REM RIGHT ICEBERG
305 REM SPRITE DATA SAME AS FOR LEFT ICEBERG
306 REM SO SPRITE POINTER #2 POINTS
307 REM TO SAME ADDRESS AS SPRITE POINTER #1
308 POKE 2042,249:REM SPRITE POINTER #2
309 REM DIRECTED TO 249*64, THAT IS 15936
380 REM ASSIGN A COLOR
381 POKE SC+2,1:REM SPRITE COLOR 2
382 REM ICEBERG APPEARS IN FRONT OF SKY
383 REM BP STILL HAS ALL ZEROS FROM LINE 183
384 REM ASSIGN INITIAL POSITION
385 POKE SS,4:REM SPRITE 2 RIGHT OF SCREEN
386 POKE X(2),16:REM 256+16=272ND COLUMN
387 REM OR 16TH COLUMN OF RIGHT SIDE
388 POKE Y(2),114:REM 114TH ROW
390 POKE EN,7:REM SET SPRITE 0,1 & 2 ON
```

For the cap on the reef, the process is the same. It won't jut above the waterline but will force the sub to surface in order to pass over it. We'll have to color it brown, of course, and enable it with a 1 in the fourth bit of EN (00001111 = 15).

```
400 REM CENTRAL REEF
405 REM SPRITE DATA SAME AS FOR LEFT ISLAND
406 REM SO SPRITE POINTER #3 POINTS
407 REM TO SAME ADDRESS AS SPRITE POINTER #1
408 POKE 2043,249:REM SPRITE POINTER #3
409 REM DIRECTED TO 249*64, THAT IS 15936
480 REM ASSIGN A COLOR
481 POKE SC+3,9:REM SPRITE COLOR 3
482 REM ISLAND APPEARS IN FRONT OF SKY
483 REM BP STILL HAS ALL ZEROS FROM LINE 183
484 REM SPRITE 3 ALREADY LEFT OF SCREEN
485 REM SS HAS 0 IN BIT 3 FROM LINE 385
486 POKE X(3),165:REM 165TH COLUMN
488 POKE Y(3),124:REM 124TH ROW
490 POKE EN,15:REM SET SPRITE-0,1,2 & 3 ON
```

Now our stage is set. SAVE and test RUN.

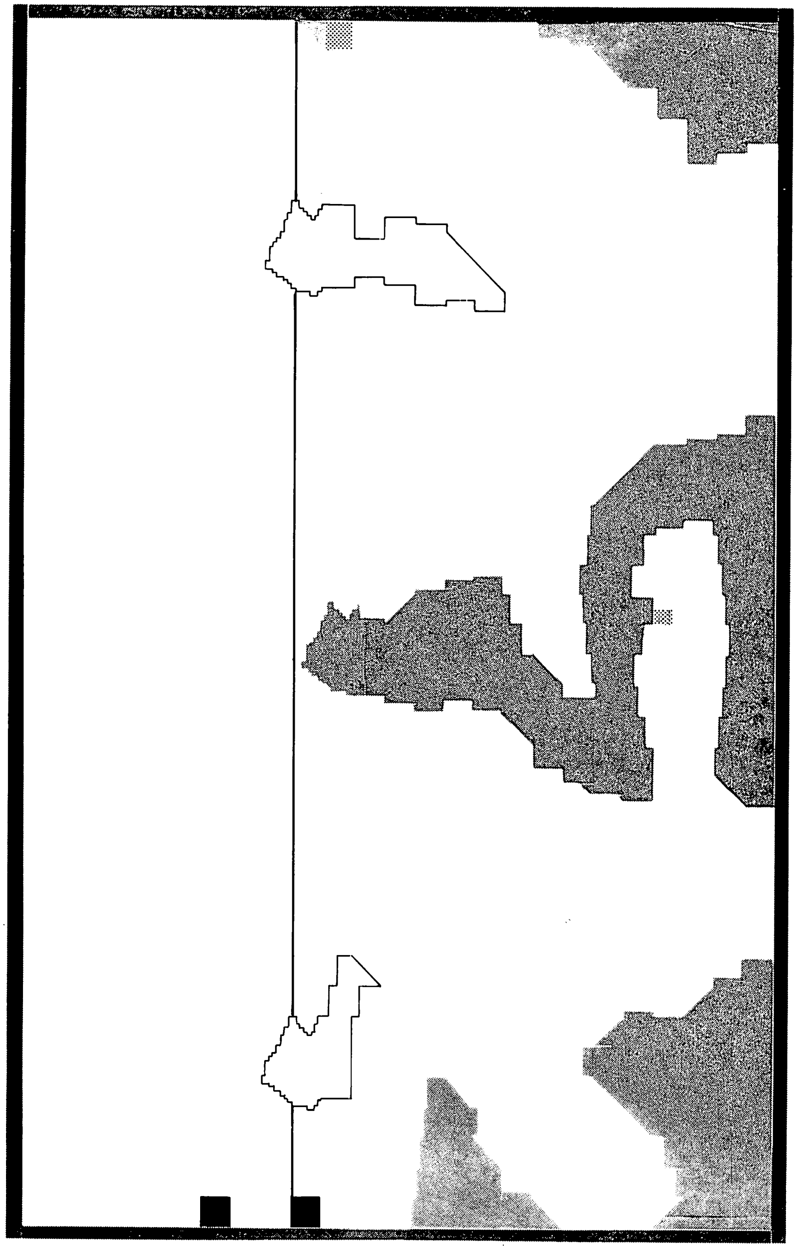


Figure 5.4 — the undersea stage

6 The action sequence

To get two or more things moving within the one program, you need a *loop* — a sequence in which first one object moves one step and then the next moves one step, and so on till you're back to the first object and ready to have it make its second step. It sounds painfully slow but the computer does it all so quickly that it appears as if all the objects are moving together!

Our program has four moving objects: the sub, the destroyer, the depth charge and the sea monster. We've got our sub moving so next on the agenda will be the destroyer.

6.1 The destroyer

For the destroyer we'll use sprite number 4. Figures 6.1, 6.2 and 6.3 show what mine looks like in pixels, binary and decimal numbers.

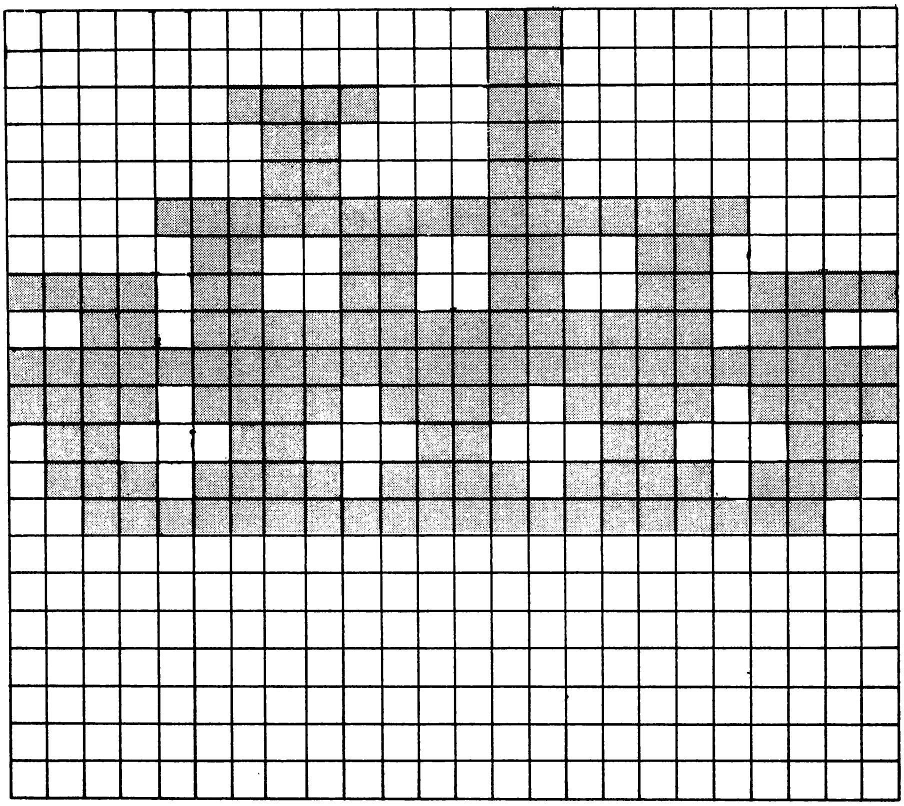


Figure 6.1 — destroyer sprite

0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	1	1	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
0	0	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	0	0	0
1	1	1	1	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	1	1	1	1
0	0	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	0	1	1	1	1	0	1	1	1	1	0	1	1	1	1	0	1	1	1	1
0	1	1	0	0	0	1	1	0	0	0	1	1	0	0	0	1	1	0	0	0	1	1	0
0	1	1	1	0	1	1	1	1	0	1	1	1	1	0	1	1	1	1	0	1	1	1	0
0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 6.2 — destroyer sprite in binary bytes

0	6	0
0	6	0
3	198	0
1	134	0
1	134	0
15	255	240
6	102	96
246	102	111
55	255	236
255	255	255
247	189	239
99	24	198
119	189	238
63	255	252
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0

Figure 6.3 — destroyer sprite in decimal bytes

And here it is as part of the program:

```
500 REM DESTROYER
505 DSMEM=16000:REM 1ST ADDRESS FOR DATA
507 POKE 2044,250:REM SPRITE POINTER #4
508 REM DIRECTED TO 250*64, THAT IS 16000
510 FOR F=0TO62
511 READ BY
512 POKE DSMEM+F,BY
513 NEXT
520 DATA0,6,0,0,6,0,3,198,0,1,134,0,1,134,0
521 DATA15,255,240,6,102,96,246,102,111
522 DATA55,255,236,255,255,255,247,189,239
```

```
523 DATA99,24,198,119,189,238,63,255,252
524 DATA0,0,0,0,0,0,0,0,0,0,0,0
525 DATA0,0,0,0,0,0,0,0,0
580 REM ASSIGN A COLOR
581 POKE SC+4,6:REM SPRITE COLOR 4
582 REM DESTROYER APPEARS IN FRONT OF SKY
583 REM BP STILL HAS ALL ZEROS FROM LINE 183
584 REM SPRITE 4 ALREADY LEFT OF SCREEN
585 REM SS HAS 0 IN BIT 4 FROM LINE 385
586 POKE X(4),204:REM 204TH COLUMN
588 POKE Y(4),108:REM 108TH ROW
590 POKE EN,31:REM SET SPRITE 0,1,2,3 & 4 ON
594 W(4)=48:E(4)=47:REM EAST & WEST LIMITS
```

SAVE and RUN. There she is, heavily armed and on the lookout for submarines.

Notice that:

- we've POKEd our ENabling byte with 31 (00011111) to put a 1 into bit number 4 of the byte
- we haven't bothered to set limits to the destroyer's vertical movements because there won't be any! The destroyer will be programmed only to move horizontally.

6.2 Random movement using a timer

Our next step will be to get the destroyer moving. To make good sport, we'll need the destroyer's movements to be unpredictable — that is, random.

If we were simply to let the destroyer jump left or right on every loop, however, it would jig back and forth in a most unboatlike fashion! We have to arrange it so that the destroyer will move in the same direction for a respectable amount of time before it reverses direction. To do this, we make use of a special byte that the computer uses as a timer. The byte is kept at addresses 56324. Try this command (without a line number):

PRINT PEEK (56324) and hit RETURN.

You get a number. Now try it again. You get a different number. In fact, the numbers in this address are changing so fast that you can use it as a source of random numbers — if you dip into it irregularly.

Numbers greater than 240 occur about once every 16 loops and that is a respectable sort of time to allow before giving the 'full astern' command. So we'll let the destroyer go on its merry way in one direction. To do this we introduce a new variable SE (for ship east). SE = 1 when the ship is going eastwards and SE = 0 when it's going westwards. The sequence is put together in lines 2255 and 2259.

When the eastern and western limits are reached, SE will have to be reversed — see lines 2266 and 2286.

```
2200 REM DESTROYER MOVEMENT
2250 REM SELECT DIRECTION
2252 PX=PEEK(X(4)):REM PRESENT X POSN
2254 SP=PEEK(SS):PS=SPAND16:REM SIDE OF SCR
2255 IF PEEK(53324)>240 THEN SE=1-SE
2259 IF SE THEN 2286:REM SE=1  SHIP GO EAST
2260 REM SPRITE WEST
2266 IF PX<W(4)ANDPS=0 THENSE=1:REM WEST LMT
2268 IF PX=0 THEN POKE X(4),252:GOTO2272
2269 REM LINE 2268 IS FOR CROSS TO WEST SIDE
2270 POKE X(4),PX-4:RETURN:REM GO WEST
2272 POKESS,SPAND239:GOTO2075:REM CHNGE SIDE
2280 REM SPRITE EAST
2286 IF PX>E(4)ANDPS THEN SE=0:REM EAST LMT
2288 IF PX=252 THEN POKE X(4),0:GOTO 2292
2289 REM LINE 2288 IS FOR CROSS TO WEST SIDE
2290 POKE X(4),PX+4:RETURN:REM GO EAST
2292 POKE SS,SPOR16:GOTO2075:REM CHANGE SIDE
```

And to fold the destroyer movement into the action loop, you'll need line 1050:

```
1050 GOSUB2252:REM DESTROYER MOVE
```

SAVE and RUN. Your destroyer should shuttle left and right unpredictably. Apart from the new variable SE, about the only difference worth noting between this movement routine and the movement routine for the sub is the number we're using with our ANDs and ORs. The number 16 is the decimal equivalent of a 1 in bit number 4 (the fifth bit from the right) of the byte.

6.3 Making a monster

Why not start right out by making your own monster? Draw him (or her) on your 24×21 grid, convert it into 63 bytes expressed in decimal numbers and enter the numbers in lines 620 and 626 instead of my DATA.

Figures 6.4, 6.5 and 6.6 show my monster — first in pixels and then in binary and decimal numbers.

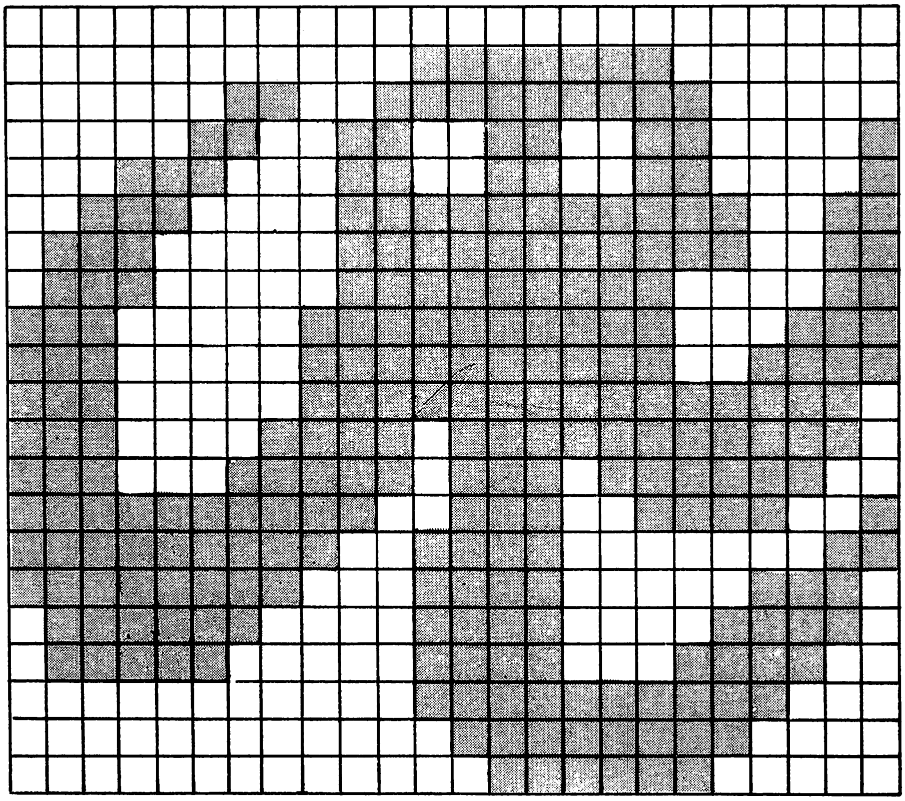


Figure 6.4 — monster sprite in pixels

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0
0	0	0	0	0	0	1	1	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0
0	0	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	0	0	1
0	0	0	1	1	1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	0	0	1
0	0	1	1	1	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1
0	1	1	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	0	0	1	1
0	1	1	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	1	1
1	1	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1
1	1	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	1	1	1	1
1	1	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
1	1	1	0	0	0	0	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	0
1	1	1	0	0	0	1	1	1	1	1	0	1	1	1	0	1	1	1	1	1	1	0	0
1	1	1	1	1	1	1	1	1	1	0	0	1	1	1	0	0	1	1	1	1	0	0	1
1	1	1	1	1	1	1	1	1	0	0	1	1	1	1	0	0	0	0	0	0	0	1	1
1	1	1	1	1	1	1	1	0	0	0	1	1	1	1	0	0	0	0	0	1	1	1	0
0	1	1	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0
0	1	1	1	1	1	0	0	0	0	0	1	1	1	1	0	0	0	1	1	1	1	0	0
0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0

Figure 6.5 — monster sprite in binary bytes

0	0	0
0	31	192
3	63	224
6	102	97
28	102	97
56	127	227
112	127	227
112	127	195
224	255	199
224	255	207
224	255	254
225	239	254
227	238	252
255	206	121
255	158	3
255	30	14
126	30	30
124	30	60
0	31	248
0	15	240
0	7	224

Figure 6.6 — monster sprite in decimal bytes

And here it is as part of the program:

```
600 REM MONSTER
605 MOMEM=16064
607 POKE 2045,251:REM SPRITE POINTER #5
608 REM DIRECTED TO 251*64, THAT IS 16064
610 FOR F=0TO62
611 READ BY
612 POKE MOMEM+F,BY
613 NEXT
620 DATA0,0,0,0,31,192,3,63,224,6,102,97
621 DATA28,102,97,56,127,227,112,127,227
```

```
622 DATA112,127,195,224,255,199,224,255,207
623 DATA224,255,254,225,239,254,227,238,252
624 DATA255,206,121,255,158,3,255,30,14,126
625 DATA30,30,124,30,60,0,31,248,0,15,240
626 DATA0,7,224
680 REM ASSIGN A COLOR
681 POKE SC+5,13:REM SPRITE COLOR 5
682 POKEBP,32:REM ONLY BIT 5 = 1
683 REM SO ONLY SPRITE 5 GOES BEHIND BKGRND
684 POKESS,36:REM SPRITES 2,5 RIGHT OF SCR
685 REM SO BITS 2&5 TOTAL 36: 4+32=36
686 POKE X(5),0:REM IST COLUMN ON RIGHT SIDE
688 POKE Y(5),242:REM 242ND ROW
690 POKE EN,63:REM SET SPRITES 0 TO 5 ON
692 U(5)=187:L(5)=242:REM UP & LOW LIMITS
694 W(5)=0:E(5)=45:REM EAST & WEST LIMITS
```

The important things to note are these:

- we're using sprite number 5
- we want our monster to appear to go *behind* the scenery (ie — the background to have priority over the sprite) so we've POKEd BP with 32, the decimal equivalent of a 1 in bit number 5 of the BP byte
- we've set quite tight limits to the monster's movement so he won't stray far from the approach to the eastern gold
- we've set bit 5 of SS permanently at 1 (in line 684) because the monster never crosses to the left sector of the screen. Similarly, bit 2 of SS for the eastern iceberg (sprite 2) always equals 1
- the ENabling byte has been POKEd with 63 so now there are 1's in all of the bits from 0 to 5 — ie sprites 0 to 5 are on

6.4. Monster movement

Our monster is a shifty fellow so we'll have to provide him with all four directions of movement and a fifth state — at rest — to make him totally lifelike.

His movements must also be unpredictable so we'll start the movement routine with a random selection from the five possibilities:

```
1300 REM MONSTER MOVEMENT
1310 MM=INT(RND(3)*5)+1
1320 ON MM GOSUB2422,2442,2462,2482,1340
1330 GOTO2542:REM GOTO MISSILE MOVE
1340 RETURN
2542 GOTO 1010
```

Line 1310 sets up the variable MM (for monster movement) and gives it the value of a set of numbers between 1 and 5 (inclusive), chosen at random.

Random numbers on the Commodore 64

As an example, consider how we make up line 1310. Let's build it up in steps:

RND(3) will randomly give us any number less than 1, for example .7362438.

RND(3)*5 will multiply the above by 5, giving us any number less than 5, for example, 3.812190.

INT(RND(3)*5) will give us any *whole number* less than 5, that is from 0 to 4.

INT(RND(3)*5)+1 will give us any whole number from 1 to 5.

(The 3 in RND(3) could be any positive number.)

In this way you can control the range of random numbers you get.

Line 1320 is a more compressed way of saying IF. . .THEN GOSUB several times. We could have said:

```
1320 IF MM=1 THEN GOSUB 2422
1321 IF MM=2 THEN GOSUB 2442
1322 IF MM=3 THEN GOSUB 2462
```

and so on for all five possibilities. ON takes the value of the variable (1 to 5) and matches it with the GOSUB line number that has the same position in the queue.

Line 1330 doesn't cause anything actually to happen yet because line 2542 hasn't been written. It's important to write line 1330 now, however, because the missile move follows next in the action sequence. Recall how we said earlier that to get two or more objects moving you had to move one first and then the other?

Line 2542 is a temporary line only. It's there so we can test RUN the sequence once we've written the actual monster movement subroutines. It closes the loop back to the sub's movement sequence.

Now for the monster movement subroutines:

```
2400 REM MONSTER MOVEMENT (= SPRITE NO. 5)
2420 REM SPRITE UP
2422 PY=PEEK(Y(5)):REM PRESENT Y POSN
2424 IF PY<U(5) THEN RETURN:REM UPPER LIMIT
2425 POKE Y(5),PY-3:RETURN:REM RAISE Y POSN
2440 REM SPRITE DOWN
2442 PY=PEEK(Y(5)):REM PRESENT Y POSN
2444 IF PY>L(5) THEN RETURN:REM LOWER LIMIT
2445 POKE Y(5),PY+1:RETURN:REM LOWER
2460 REM SPRITE WEST
2462 PX=PEEK(X(5)):REM PRESENT X POSN
2466 IF PX=W(5) THEN RETURN:REM WEST LIMIT
2470 POKE X(5),PX-3:RETURN:REM GO WEST
2480 REM SPRITE EAST
2482 PX=PEEK(X(5)):REM PRESENT X POSN
2486 IF PX=E(5) THEN RETURN:REM EAST LIMIT
2490 POKE X(5),PX+3:RETURN:REM GO EAST
```

SAVE and RUN. Do you notice an irregularity in the monster's movement? Yes, it takes three steps up to one step down. The effect of this is to make the monster float upwards to menace the approaching sub. Can you see the lines that create this pattern? They are the lines that raise and lower the sprite.

6.5 Creating the missile

The depth charge (or missile) is another sprite. It uses only eight pixels and consists of two narrow black bands, to represent two depth charges dropped in quick succession. Most of the DATA for this sprite consists of 0's but we still have to write all of them in.

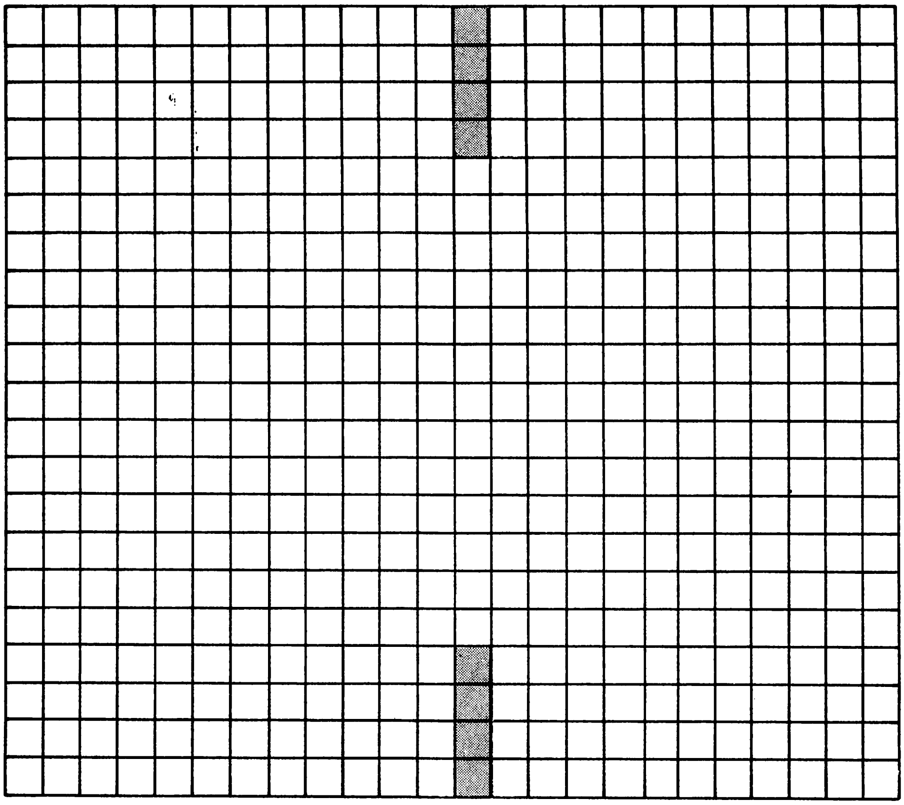


Figure 6.7 — missile sprite in pixels

0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0

Figure 6.8 — missile sprite in binary bytes

0	8	0
0	8	0
0	8	0
0	8	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	8	0
0	8	0
0	8	0
0	8	0

Figure 6.9 — missile sprite in decimal bytes

```
700 REM MISSILE
705 MISMEM=16128
707 POKE 2046,252:REM SPRITE POINTER #6
708 REM DIRECTED TO 252*64, THAT IS 16128
710 FOR F=0TO62
711 READ BY
712 POKE MISMEM+F,BY
713 NEXT
720 DATA0,8,0,0,8,0,0,8,0,0,8,0
721 DATA0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
722 DATA0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
723 DATA0,0,0,0,0,0,0,8,0,0,8,0,0,8,0,0,8,0
780 REM ASSIGN A COLOR
```

```
781 POKE SC+6,0:REM SPRITE COLOR 6
782 POKEBP,96:REM ONLY BITS 5 & 6 = 1
783 REM SPRITES 5 & 6 GO BEHIND BACKGROUND
784 REM SPRITE 6 LEFT OF SCREEN
785 REM SS HAS 0 IN BIT 6 FROM LINE 385
786 POKE X(6),204:REM DESTROYERS INIT COLUMN
788 POKE Y(6),123:REM 16 ROWS UNDER DEST ROW
790 POKE EN,127:REM SET SPRITES 0-6 ON
792 L(6)=206:REM JUST ABOVE SUB ON MUD
794 REM E & W LMTS CONTROLLED BY DEST LMTS
```

There are five points to note about sprite number 6:

- there are no east or west limits because the missile column is dictated by the destroyer column at the time of the missile's release
- missiles pass behind scenery so a 1 has to be placed in bit number 6 of the byte for background priority (BP). A 1 in bit number 6 = 64 but recalling that sprite number 5 also passes behind the scenery, we're reminded that we have to POKE BP with 32 + 64 = 96
- the sprite's initial Y position is 16 rows (not 21) below the destroyer's initial Y position because the destroyer doesn't take up the whole 21 rows of its pixel grid. (Note also that both missiles appear on the screen simultaneously and move in tandem down the screen because they are part and parcel of the same sprite)
- the ENabling byte, POKEd with 127, now looks like this: 01111111
- line 792 sets the critical depth at which missiles explode — if they've not hit anything else on the way.

6.6 Missile movement

Once released, of course, our missile is obliged to drop one step in every loop until it hits something or reaches the depth where it explodes automatically.

```
2500 REM MISSILE MOVEMENT (= SPRITE NO. 6)
2540 REM SPRITE DOWN
2542 PY=PEEK(Y(6)):REM PRESENT Y POSN
2544 IF PY>L(6) THEN 2700:REM LOWER LIMIT
2545 POKE Y(6),PY+6:REM LOWER
2700 REM MISSILE EXPLODE
```

```
2750 POKE X(6),PEEK(X(4)):REM DESTR COLUMN
2755 PS=PEEK(SS):POKE SS,PSAND191
2756 IF PSAND16 THEN POKE SS,PSOR64
2760 POKE Y(6),123:REM 16 ROWS BELOW DESTR
2780 GOTO 1010:REM BACK TO SUB MOVE
```

You can SAVE your program now and give it a test RUN but there won't be any actual explosion because we haven't written the explosion lines yet. The missile simply disappears when it reaches the lower limit we set in line 792.

Line 2544 directs the program to the explosion routine (which we'll write later) starting at line 2700 IF the missile's depth is greater than L(6).

Line 2545 lowers the missile through the water, +6 rows at a time.

Lines 2750 to 2760 handle the work of keeping the missiles coming in a constant flow. As soon as one missile has exploded, we'll start dropping another. Before we lose our missile in the explosion, then, we'll set the new column and row positions for X(6) and Y(6), our missile variables.

Line 2750 changes the missile column value, X(6), to the destroyer's present column value, X(4).

Line 2755 ensures that the new missile appears on the correct side of the screen. In binary notation, PS AND 191 = PS AND 10111111. This gets rid of the 1 from bit 6. The new missile is now set at left of screen.

Line 2756 checks whether the destroyer is right of screen and if it is, inserts a 1 at bit 6 of SS — putting the new missile also at right of screen.

Line 2760 assigns the missile row variable, Y(6), a value 16 more than the destroyer's row. This makes the missile appear to have just been dropped.

7 Collision detection

At the moment, submarines can drive through rocks and missiles pass harmlessly through submarines! This state of affairs cannot be allowed to continue so our next step will be to call the computer's attention to these collisions. One of the great features of the C-64 sprite system is automatic collision detection. Let's see how it works.

7.1 Two types of collision

There are two possible sorts of collision:

- sprite with sprite and
- sprite with background.

The collisions are recorded at two separate addresses — ie there is one PEEK for each sort. For sprite/sprite collisions we PEEK 53278 and for sprite/background collisions we PEEK 53279.

To streamline the process, these addresses automatically reset to zero after they've been PEEKed. So let's PEEK them at the start to clear them.

```
1005 F=PEEK(53278):F=PEEK(53279)
1006 REM ZERO COLLISION CHK (1ST TIME ONLY)
```

7.2 Submarine collision

Our sub is sprite 0 so we check the 0 bits of the bytes at addresses 53278 and 53279 to determine whether there's been a collision. Naturally, our collision detector won't produce any *boom* yet because we haven't written the sequence for it. All it will do is set CO (for collision) to 1. Later on, when we make the explosions, CO = 1 will be the trigger.

```
2100 REM SUB COLLISION CHECK
2110 REM COLLIDE WITH BACKGROUND?
2112 PX=PEEK(X(0))
2115 PY=PEEK(Y(0))
2120 IF PEEK(53279)AND1 THEN CO=1
2130 REM COLLIDE WITH OTHER SPRITE?
2140 IF PEEK(53278)AND1 THEN CO=1
```

In lines 2120 and 2140 the AND 1 checks the zero bit of the byte to see whether it contains a 1.

If our sub has surfaced, though, we'll have to arrange it so that the collision detection with the background will be avoided. Recall that the whole sky is background because it's made up of reverse spaces and they are actually keyboard characters. So we'll avoid background collision detection when PY is less than 123 (the row where the water starts) and write the jump at the end of line 2115:

```
2115 PY=PEEK(Y(0)):IF PY<123 THEN 2140
```

Then to reset the collision detector for sprite with background collisions (it will have recorded the collision with the sky even though we don't want to act on the information), we'll need:

```
2145 F=PEEK(53279)
2150 RETURN
```

Note that we also have to reset the collision PEEKs to zero when we change side of screen. This is because changes to side of screen cause column values suddenly to shift to a different screen location. The result can be alarming: clearwater collisions where the hazards are to be found only in your computer's memory!

Left and right movements for the sub always exit through line 2075 so we can include the clearing PEEKs there:

```
2075 F=PEEK(53278):F=PEEK(53279):RETURN
```

And finally, to fold the collision check into the submarine control routine, we'll need:

```
1040 GOSUB2100:REM COLLISION CHECK
```

7.3 Missile collision

In the case of the missile, we only have to deal with collisions with the background. To be sure, we want the misile to explode when it hits the sub as well as when it hits the background but we don't have to do anything about it: we've already catered for submarine hitting missile!

Nor will we allow the missile to blow up our sea monster. Naturally, sea monsters are smart enough to dodge any suspicious looking objects that come their way in the deeps.

Our missile is sprite number 6 so we check bit number 6 of the byte at address 53279:

```
2600 REM MISSILE COLLISION CHECK
2610 REM COLLIDE WITH BACKGROUND?
2620 CM=0:IF PEEK(53279)AND64 THEN CM=1
2650 IF CM=0 THEN 1010:REM BACK TO SUB MOVE
2700 GOTO 1010
```

This sequence won't produce any explosion yet either, because the explosion lines haven't been written. It will simply set CM (for missile collision) to 1. Later on, when we make the explosion, CM = 1 will be the trigger.

In line 2620 the AND 64 checks for a 1 in bit number 6 of the byte.

Line 2700 is only temporary so you can test RUN your program so far. Later on, 2700 will be the first line of the missile explode routine.

8 Explosions

The design of our explosion sprite provides us with a good opportunity to try out the C-64's ability to produce multi-color sprites.

The C-64 will grant us this greater variety of color in our sprite if we are willing to use pixels twice as long as the regular square pixel. That means that we'll have slightly less choice about the exact shape or outline of our sprite object.

With an explosion we can be fairly free with our design so the limitations of having to use chunky pixels won't worry us much.

8.1 The multi-color pixel grid

Figure 8.1 shows the multi-color pixel grid (in case you want to design your own explosion sprite). Notice that each byte of this grid contains only four *double length* pixels. The grid is still twenty-one bricks deep but is now only twelve (double length) bricks wide.

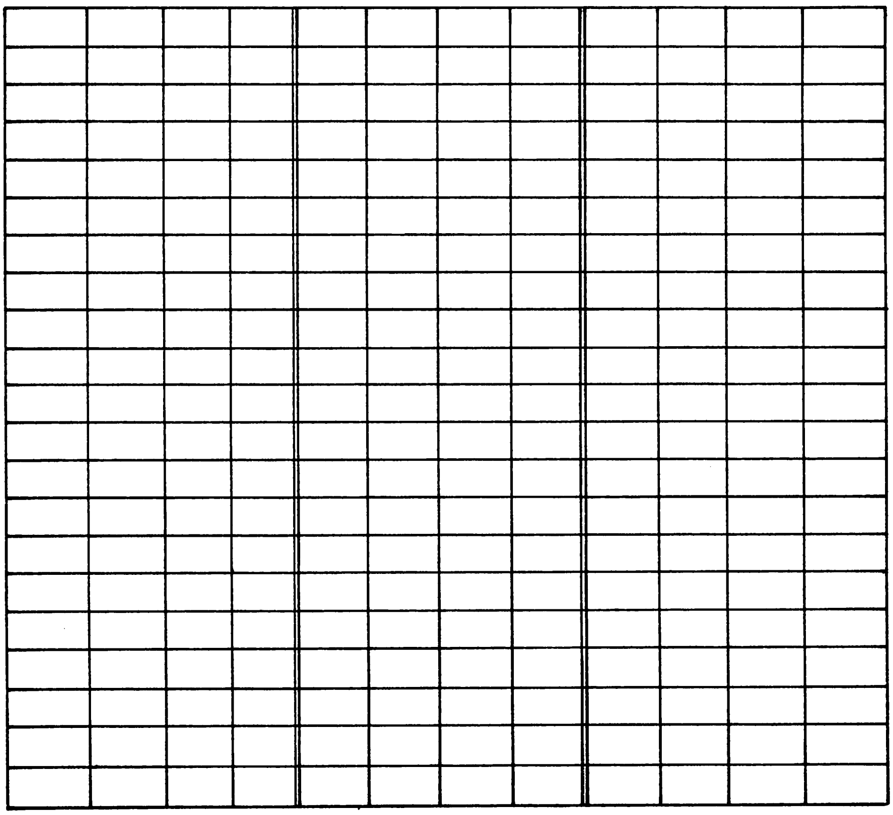


Figure 8.1 — the multi-color sprite grid

8.2 Coding the colors

With any multi-colored sprite, we have the choice of four colors:

- screen color (virtually invisible)
- multi-color 0
- multi-color 1
- regular sprite color

The 'multi-colors', of course, are just one color each. Once they are set, these multi-color choices remain the same for all multi-colored sprites. The regular sprite color, however, can be different for each sprite.

The regular sprite color is set in the usual way but the multi-colors have their own addresses into which you POKE the code number of the color you want. The address for multi-color 0 is 53285 and the address for multi-color 1 is 53286.

Now let's say I have a byte that I want to color like this:

RED	WHITE	YELLOW	INVISIBLE
Sprite color	multi-col 0	multi-col 1	screen

We've got only four pixels to the byte. But we still use up the full eight bits in the byte telling the computer both *where* we want the pixels and *what color* we want them to be. To do that, we code the pixels according to this list:

- 00 for screen color
- 01 for multi-color 0
- 11 for multi-color 1
- 10 for regular sprite color

So, in its new code, my byte will now look like this:

10	01	11	00

Putting the pairs of numbers into one long line gives the binary number 10011100. And in decimal notation, that's 156 (128+16+8+4). So to code any object you've mapped out on a multi-color grid, you:

- mark up the grid with the colors you've chosen for multi-color 0, multi-color 1 and regular sprite color
- convert the colors (and blanks) to their two digit codes
- read each set of four pairs as an eight digit binary number and convert it to its decimal equivalent
- POKE 53285 with the regular color code for multi-color 0.
- POKE 53286 with the regular color code for multi-color 1.

Following that sequence, Figures 8.2, 8.3 and 8.4 show my design for the multi-colored explosion sprite.

						Y					Y
						Y					Y
R						Y				Y	
Y	R					Y			Y	R	
	Y	R				R		Y	R		
	R	Y				R		R			
		R	Y	R	Y	Y	R	Y			
			R	Y	W	W	Y				
				R	W	W	Y	Y			
		R	R	Y		R	W	Y	Y		
Y	R	R	W	W	W	W	Y	Y			
			Y	W	W	W	Y				
			Y	Y	W	W	R	Y			
			R	R	R	W	W	R			
		Y	R		R	Y	Y	Y	R		
		R	R		R				Y	R	
	R	Y			R					Y	R
	Y				R			R			Y
Y					R			Y			
Y					Y				Y		

Figure 8.2 — explosion color scheme

R = Red Y = Yellow W = White

00	00	00	00	00	00	00	00	00	00	00	00
00	00	00	00	00	00	11	00	00	00	00	11
00	00	00	00	00	00	11	00	00	00	00	11
10	00	00	00	00	00	11	00	00	00	11	00
11	10	00	00	00	00	11	00	00	11	10	00
00	11	10	00	00	00	10	00	11	10	00	00
00	10	11	00	00	00	10	00	10	00	00	00
00	00	10	11	10	11	11	10	11	00	00	00
00	00	00	10	11	01	01	11	00	00	00	00
00	00	00	00	10	01	01	11	11	00	00	00
00	00	10	10	11	00	10	01	11	11	00	00
11	10	10	01	01	01	01	11	11	00	00	00
00	00	00	11	01	01	01	11	00	00	00	00
00	00	00	11	11	01	01	10	11	00	00	00
00	00	00	10	10	10	01	01	10	00	00	00
00	00	11	10	00	10	11	11	11	10	00	00
00	00	10	10	00	10	00	00	00	11	10	00
00	10	11	00	00	10	00	00	00	00	11	10
00	11	00	00	00	10	00	00	10	00	00	11
11	00	00	00	00	10	00	00	11	00	00	00
11	00	00	00	00	11	00	00	00	11	00	00

Figure 8.3 — explosion in binary bytes

0	0	0
0	12	3
0	12	3
128	12	12
224	12	56
56	8	224
44	8	128
11	190	192
2	215	0
0	151	192
10	213	240
233	87	192
3	87	0
3	214	192
2	165	128
14	47	224
10	32	56
44	32	14
48	32	131
192	32	192
192	48	48

Figure 8.4 — explosion in decimal bytes

And here it is as part of the program:

```
800 REM EXPLOSION
805 EXMEM=16192
810 FOR F=0TO62
811 READ BY
812 POKE EXMEM+F,BY
813 NEXT
820 DATA0,0,0,0,12,3,0,12,3,128,12,12,224,12
821 DATA56,56,8,224,44,8,128,11,190,192
822 DATA2,215,0,0,151,192,10,213,240,233,87
823 DATA192,3,87,0,3,214,192,2,165,128
824 DATA14,47,224,10,32,56,44,32,14,48,32
```

```
825 DATA131,192,32,192,192,48,48
880 REM ASSIGN A COLOR
882 POKE 53285,1:REM MULTI COLOR 0
883 POKE 53286,7:REM MULTI COLOR 1
884 REM 3RD COLOR = SPRITE 6 COLOR (BLACK)
885 REM WILL BE MADE RED DURING EXPLOSION
```

Notice that we haven't actually set the regular sprite color for the explosion. Lines 884 and 885 speak to us but not to the computer. We haven't set the regular sprite color because we're actually going to *replace* (temporarily) the missile sprite with the DATA for the explosion sprite, when the missile hits something. So setting the regular sprite color for the explosion graphic will actually involve changing the sprite color of sprite number 6 — the missile sprite.

8.3 Missile explosion

When the missile hits something, we want our brightly colored explosion sprite to be displayed in exactly the last position occupied by the missile. The best way to do this is simply to give the missile sprite the explosion DATA, at its point of impact:

```
2700 REM MISSILE EXPLODE
2720 POKE 2046,253:REM MISSILE POINTER
```

And recall that we were going to set the color for sprite 6 to red for the explosion:

```
2717 POKE SC+6,2:REM SPRITE COLOR 6
```

Some background reflection from the explosion would also be appropriate so we'll set the screen color to purple:

```
2710 POKE 53281,4:REM SCREEN COLOR
```

Naturally, however, we'll have to set it back again when the flash has subsided:

```
2725 FOR F=1TO100:NEXT
2730 POKE 53281,14:POKE53280,9:REM SCR COL
```

And we'll also have to restore the regular color to our missile and the pointer to its proper missile DATA:

```
2735 POKE SC+6,0:REM SPRITE COLOR 6
2740 POKE 2046,252:REM MISSILE PTR TO NORMAL
```

All of this looks promising but nothing will happen unless we tell the computer to read sprite 6 in multi-color mode. The byte that holds this information is held at address 53276. It's one of those bytes that serves all eight sprites with a bit for each sprite. To put a 1 into bit number 6 we POKE it with 64:

```
2715 POKE 53276,64:REM SPR 6 MULTI COL MODE
```

And when the display is complete, we reset it to its regular, high resolution mode:

```
2745 POKE 53276,0:REM ALL SPRS TO HIRES MODE
```

The explosion looks messy as it changes back to the missile graphic so the best thing to do is simply turn it off until the next missile drop starts:

```
2726 POKE EN,63:REM DISABLE SPRITE 6
2727 REM ENABLE ONLY SPRITES 0-5
```

We then re-ENable it after the new missile starts:

```
2765 POKE EN,127:REM RE-ENABLE SPRITES 0-6
```

SAVE and RUN. You should see multicolored flashes whenever the missile makes impact!

8.4 Submarine explosion

When the submarine explodes, of course, the display should be much more dramatic than the missile exploding against rocks or spending its energies harmlessly in the deep. For the submarine explosion we'll use the same explosion graphic but we'll cause it to expand to double its original size. This is another of the remarkable features of sprite graphics on your C-64!

A submarine explosion becomes appropriate after a collision check and IF CO = 1. (Saying IF CO is the same as saying IF CO = 1.)

```
1045 IF CO THEN 2800:REM CRASH
```

The appropriate place for the explosion to occur is the sub's last position so we'll simply change the sub into our explosion. We just have to set the sub's pointer to the explosion DATA.

```
2800 REM SUB EXPLODE
2820 POKE 2040,253:REM SUB POINTER
2821 REM TEMP REDIRECTED TO BOOM SPRITE
```

We'll also create a heat flash (screen color change) and switch to multi-color mode:

```
2810 POKE 53281,4:POKE53280,0:REM SCR COLOR
2815 POKE 53276,1:REM SPR 0 - MUL COLOR MODE
```

And let's not forget that we'll want red as our sprite color:

```
2817 POKE SC+0,2:REM SPRITE COLOR 0
```

Then, after a short pause:

```
2825 FOR F=1TO200:NEXT
```

we'll expand the sprite to double its original size. To do this we have to use two separate commands — one to expand it along its X axis and one to expand it along its Y axis. The bytes that hold this information are held at addresses 53277 and 53278 respectively. Again, they are those sorts of bytes that serve all eight sprites with a bit for each sprite. To widen the sprite, then, we POKE a 1 into bit number 0 of the byte at 53277:

```
2827 POKE 53277,1:REM POKE SPRITE WIDENER
2828 REM WITH BIT 0 ON (FOR SPRITE 0)
```

and to deepen it, we POKE a 1 into bit number 0 of the byte at 53278 (with our usual pauses so it all doesn't disappear too quickly):

```
2830 FOR F=1TO200:NEXT
2835 POKE 53271,1:REM POKE SPRITE DEEPENER
2836 REM WITH BIT 0 ON (FOR SPRITE 0)
2837 FOR F=1TO200:NEXT
```

And after all this excitement, we restore the screen to its regular color and print the fateful news:

```
2845 POKE 53281,14:POKE53280,9
2846 REM SCREEN COLOR BACK TO NORMAL
2850 PRINT"[illegible]SUB DESTROYED"
2860 STOP
```

SAVE and RUN. You should see a dramatic explosion resulting from bad driving, copping a depth charge, or getting mangled by the monster!

9 Sound effects

Now for the sound effects. We'll need *booms* to accompany the explosions and a *sonar ping* to enhance the release of a missile from the hunter-destroyer.

9.1 The sound addresses

First of all, to save ourselves a lot of typing later on, we'll set up some abbreviations for the sound addresses. Naturally, they are best located early on in the program so they'll always be available.

Your C-64's sound synthesiser has three 'voices' (sound channels) and each voice has its own set of addresses for the elements needed to make it work. In this program we're going to use two of the three voices.

```
70 REM SOUND ABBREVIATIONS
72 VO=54296:PI=54273:DE=54277:GA=54276
73 REM VOLUME,PITCH,DECAY,GATE ADDRS VOICE 1
74 P2=54280:D2=54284:G2=54283
75 REM PITCH,DECAY & GATE ADDRS FOR VOICE 2
76 REM VOICE 2 USED IN SUB CRASH
78 POKE VO,5:REM VOLUME SETTING FOR SONAR
79 POKE GA,0:POKE G2,0:REM CLEAR SOUND GATES
```

Line 72 sets up the addresses for the first voice. VOlume and PItch refer to the loudness and highness of the sound. DEcay refers to the way a sound fades away. GAte starts and stops the sound and also controls the waveform of the sound we want.

Line 74 sets up the addresses for voice two. Notice that we only need addresses for pitch, decay and gate (P2, D2 and G2). The VOlume address serves both voices.

Line 78 creates the volume setting for the sonar ping. There are 15 settings to choose from, where 15 is the loudest. We start out with it at the level we want for the ping because it's the sound that will occur first.

Line 79 clears (ie turns off) the sound gates for both voices. It may sound strange, but before you can turn a gate on, you have to turn it off! That's because a voice won't accept a new sound unless the old sound is completed. To be on the safe side, it's always best to start the sound sequence of a program by clearing the gates you want to use.

9.2 Sonar pings

A sonar ping is high-pitched sound of relatively short duration. We'll fold it in with our falling missile:

```
2560 REM SONAR PING
2562 POKE DE,9
2564 POKE PI,200:POKE GA,33
2565 REM SET HI PI & WAVEFORM 32 (+1 FOR ON)
2566 FOR F=1TO20:NEXT
2568 POKE GA,0:REM CEASE SOUND
```

Line 2562 sets the decay level to 9. There are 15 decay settings to choose from where 15 is the longest.

Line 2564 sets the pitch and turns the gate on with the setting for the particular waveform we want. There are a possible 253 settings for pitch, so 200 is a high sound. The gate value, 33, is actually read by the computer as 32 + 1. The number 32 is for the waveform (it's a brassy sound) and the number 1 is the signal to turn the gate on.

Line 2566 sets the duration of the sound.

Line 2568 turns the sound off and clears the gate so we can initiate subsequent sounds.

Waveform settings for GA and G1

Triangle	Sawtooth	Pulse	Noise
16	32	64	128

Adding 1 to any of these numbers opens the GAte at the same time as it sets the desired waveform.

9.3 Missile boom

This is the sound the missile makes when it explodes against a rock or goes off harmlessly at its critical depth. We want to save the most dramatic of our sound effects for the demise of our sub, so let's not be too extreme in the settings we choose for this minor explosion.

```
2705 POKEVO,10:POKEDE,11:POKEPI,3:POKEGA,129
2706 REM LOW PITCH & NOISE GENERATOR
2770 POKE GA,0:REM KILL SOUND
```

We've given it a volume setting of 10 (there's still room to make a bigger boom later) and a decay of 11. This will cause the sound to fade away naturally. The pitch, of course, is very low (3) and the waveform is for noise.

9.4 Submarine boom

Here we write a story similar to the missile boom but louder and with longer decay. This is also where we bring in our second voice to add strength to the sound. The settings are the same for the second voice but we don't have to repeat the volume setting. The one setting serves all the voices.

```
2805 POKEDE,14:POKEVO,15:POKEPI,3:POKEGA,129
2806 REM LOW PITCH & NOISE GENERATOR
2807 POKED2,14:POKEP2,3:POKEG2,129
2808 REM LO PI & NOISE GENERATOR ON VOICE 2
```

SAVE and RUN. Adjust the volume control on your TV monitor to a level comfortable for the people in the next room!

10 Frills

10.1 Scoring the game

The first question we've got to concern ourselves with is the nature of the game itself. What is the challenge? How can one player beat another? What constitutes 'doing better'? Well, apart from coming out of it alive, the shorter the time it takes you to bring the gold back to base, the better sub commander you are.

One way to score the game, then, would be to give a readout of the actual time it took a player to complete the task, without imposing any limits on the total time allowed. Rather than do it that way, however, we're going to employ a device that will really stimulate the adrenalin as the game progresses and put a limit on the time allowed.

10.1.1 The crucial counter

The limit on the time allowed to complete the mission will be the sub's fuel supply. A *fuel count* will be our *real time* indicator and will tick away relentlessly from the moment the game starts. We'll start with 2500 litres:

```
60 REM FUEL SUPPLY
65 LT=2500
```

and for every loop of the action sequence, we'll reduce it by 5:

```
1012 LT=LT-5:IF LT=0 THEN 3000
```

When you've run out of fuel, of course, the game is over:

```
3000 REM FINISH
```

But to let you know how you're going (and keep up the tension), we'll print the current state of the fuel supply at the top of the screen:

```
1014 PRINT"■■FUEL:■"LT"■ "
```

To get the symbols in line 1014 you press:

- CLR/HOME (for top of screen)
- BLACK (for the word FUEL)
- WHITE (for the number of LT's left)
- CRSR LEFT — and a space

We need the CRSR LEFT and space at the end of the line to avoid the display of the final zero when the LT count drops from 1000 to 995. Without it, the final zero of the 1000 would be left showing.

10.1.2 The remarks of fate

When line 3000 is reached (the start of the FINISH sequence), we check to see whether you are with or without fuel. If you're at this point in the program and you still have fuel (ie IF LT) THEN you must have returned to base:

```
3002 IF LT THEN 3010
3010 PRINT"■■■■■■■■■■■■■■RETURNED TO BASE"
```

If you're out of fuel, your sub won't blow up but the game will nevertheless be over. We'll arrange it so that you get a reasonably friendly message and GOTO the line that will display the previous high score:

```
3005 PRINT"■■■■■■■■■■■■■■■OUT OF FUEL"
3006 PRINT"■■■RADIO FOR HELP":GOTO 3052
```

For the successful commanders, though, we'll give points (PT) for getting the gold and points for fuel saved (LT). The bottom gold (BG) will be worth 800 points and the top gold (TG) will be worth 2000 points. We'll also give a bonus for getting both lots. The bonus will bring the possible 2800 points for the gold up to 4000 points!

```
3020 PT=300*BG+2000*TG
3025 IF PT=2800 THEN PT=4000
3027 IF PT=0 THEN PRINT"▓▓EMPTY HANDED"
3028 IF PT=0 THEN PRINT"▓▓NO SCORE":GOTO3052
3030 IF BG THEN PRINT"▓▓WITH GOLD FROM DEEP"
3035 IF TG THEN PRINT"▓▓WITH GOLD FROM EAST"
3040 PRINT"▓▓FUEL SAVED ="LT"LITRES"
3045 PT=PT+LT
3050 PRINT"▓▓YOU SCORED"PT"POINTS"
```

10.1.3 Remembering the highest score

It's always useful if the computer can recall the previous best score achieved on the game. It solves disputes and gives a player something against which to compare his or her score.

The problem about storing a previous score under a regular variable name, though, is that every time we RUN a program from the start, we lose the values of all our variables. Restarting a program clears the values stored under the variable names. Fortunately, there's a solution to the problem. The solution is to store the old high score in an address that is outside the normal program storage area in the computer's memory. Such an address is 49152. A number stored in address 49152 will stay there as long as the computer is switched on. It clears, of course, when you switch the computer off.

Here, then, is how we can store the HIgh score between RUNs.

```
3055 PRINT"▓▓PREVIOUS RECORD ="HI
3060 IF PT>HI THEN POKE 49152,PT/20
```

Line 3060 says that IF PT (the current point score) is greater than HI (the previous best) THEN to POKE 49152 with *one twentieth* of the current score. We're only storing one twentieth of the score because we're dealing with such large numbers. In most cases (we hope) the score will be greater than 255, the maximum any address can hold.

This clever storage trick, though, means that our old HI score will always be an even multiple of twenty — the computer automatically 'rounds down' the result of a division before it stores the result in a byte. If we stored, say, 4015 we'd get back 4000 when we asked the computer to PRINT PEEK (49152).

To avoid arguments and protests, then, we'll also round off every score to the nearest 20 below before we display it on the screen. To do that, we'll add a second part to line 3045:

```
3045 PT=PT+LT:PT=20*INT(PT/20)
```

Line 3045 now says that PT equals 20 times the whole number (INT) which results from dividing PT by 20.

Now to get the previous record score so that it can be printed in line 3055, we PEEK the address into which we POKEd that score in some prior game — not forgetting that we've got to multiply it by 20 to get the correct number:

```
3052 HI=PEEK(49152)*20
```

Note: when the commander has crashed his craft, he gets no points but we'll still post up the old high score:

```
2860 GOTO 3052
```

To avoid the possibility of getting a 'phantom' high score reading on your first game, it's a good idea to zero address 49152 for the start of the first game:

```
40 REM CLEAR HI SCORE STORAGE ADDRESS
45 POKE 49152,0
```

10.2 A handy restart

A nice feature of any game is a handy restart facility — something that will save the player having to type in the whole re-run command.

```
3070 PRINT"██PRESS BAR FOR MORE"
3075 GET A$:IF A$<>"" THEN 3075
```

```
3078 GET A$:IF A$<>" " THEN 3078
3079 POKE GA,0:POKE G2,0:POKE 53277,0
3080 REM KILL SOUND
3081 POKE53277,0:POKE 53271,0
3082 REM KILL WIDEN & DEEPEN
3083 POKE 53269,0:REM DISABLE ALL SPRITES
3084 POKE 53276,0:REM ALL SPRS TO HIRES MODE
3085 RUN 47:REM RESTART
```

Line 3070 simply displays the invitation PRESS BAR FOR MORE.

Line 3075 gets rid of 'bar bumps' that are stored in the keyboard buffer as a result of tense play. Without this sort of line, the computer often restarts the game before you are ready!

Line 3078 uses the GET command to assign the next key press to the variable A$. It then instructs the computer to move back to the start of line 3078 IF the value of A$ is not a bar press (the space between the quotes). That means that if bar press *is* the value of A$, the computer will pass on to line 3079 and eventually to line 3085, the line that restarts the program. If you don't press the bar, the program will simply sit at line 3070 displaying PRESS BAR FOR MORE until you stop the program with your RUN/STOP key.

Lines 3079 to 3084 reset the sound and sprite controls.

In line 3085, the command RUN automatically performs the important task of resetting to zero all of the variables you've used throughout the program. Notice, however, that we've told the computer only to RUN the program from line 47 onwards. That's so that it misses the instruction to clear the address for storage of high scores that we wrote at line 45.

10.3 Instruction display

The final professional touch is to display a set of user-friendly instructions for those who are new to the game. We're writing them last but, of course, they are displayed before the game starts.

```
  47 GOTO 4000:REM INSTRUCTIONS
4000 REM INSTRUCTIONS
4010 POKE 53280,0:POKE 53281,0:PRINT"♥";
4015 PRINT"▥          **** SUB COMMANDO ****"
4020 PRINT"▓▓YOU ARE AT WAR WITH";
```

```
4021 PRINT" THE IMPERIAL FORCES.";
4022 PRINT"YOUR GOLD IS HIDDEN UNDER ";
4023 PRINT"THE SEA.":PRINT"IT IS NEEDED ";
4024 PRINT"TO CONTINUE THE WAR EFFORT.";
4025 PRINT"▓╥THERE ARE TWO CACHES OF GOLD -"
4026 PRINT"ONE IN THE DEEP ";
4027 PRINT"AND ONE IN THE EAST."
4030 PRINT"▓▟AS A COMMANDO ";
4031 PRINT"YOU ARE EXPECTED TO"
4035 PRINT"RISK ALL TO RETRIEVE THE GOLD."
4040 PRINT"▓◣A DESTROYER PATROLS THE AREA.";
4041 PRINT" ITS DEPTH CHARGES EXPLODE ";
4042 PRINT"BEFORE REACHING BOTTOM."
4050 PRINT"▓▦AVOID THE SEA ";
4051 PRINT"MONSTER THAT GUARDS THE"
4055 PRINT"ENTRANCE TO THE EASTERN GOLD.▓"
4060 PRINT"▒YOUR ORDERS ARE TO RETURN WITH";
4061 PRINT" THE GOLD,SURFACE AT WEST BASE";
4064 PRINT" (GREY) AND ENTER";
4065 PRINT"   THE BASE (GO OFF SCREEN)"
4066 PRINT"╥▙_____________________________________";
4067 PRINT"_____________";
4068 PRINT"▩ CONTROLS: A-LEFT";
4069 PRINT" D-RIGHT F5-UP F7-DOWN "
4070 PRINT"◣▟▐▐▐▐▐▐▐▐▐▐▐PRESS BAR TO START▜";
4080 GET A$:IF A$<>" " THEN 4080
4085 RUN 50:REM START
```

Happy hunting!

11 How to write your own computer game

11.1 Variations on a theme

With a little imagination and the programming skills you've learned in the creation of *Commando Sub,* you can create a host of other, entertaining games.

For example, if you color the sky black, convert the sea into atmosphere, the rocks into unusually shaped alien dwellings and the submarine to a spaceship, you'll have a *Planetary Raider.* And if you dot the sky with planets or meteors you'll have some hazards that will really test your flying skills!

In the same way, you can convert *Commando Sub* to a land-based maze adventure, using a military craft whose mission it is to explore the territory in search of terrorists. You would expand the background to cover the whole screen. Terrorists are no easy target because they are armed with grenades and can hide behind rocks. For this version the viewer is looking *down* on the screen scene.

You can make the games you create as hard or as easy as you like, depending on the skills of the players. You can increase or decrease the speed at which the craft moves. You can add a variety of different hazards and features that will score bonus points.

It's all a matter of imagination. Imagination to change the *interpretation* of the symbols and their movements and imagination to create the sound and graphic effects that make these games so exciting. You'll be surprised at how little of the basic program you'll have to change to create something really different.

11.2 Creating a new game

If you're creating a new game you'll find it a lot easier if you plan it thoroughly first. The more time you spend in planning, the smoother your programming will flow. Here's a good sequence to follow:

11.2.1 Write the story

Write (in English) the story line of your game. Keep it really simple to start with. You can add the enhancements later on.

11.2.2 Break it into modules

- Identify the characters in your story — ie the objects that will perform some action. List them and note what you want each of them to do.
- Describe the background against which the action will take place. Break it up into various 'fields' — eg, ground and sky, 'dangerous' and 'safe' areas — and note the items you're going to put into these fields. A rough sketch is a useful way to do it.
- Decide how the game will be scored — what's really the *object* of the game?
- Note how you want the game to end.
- Design your sprites, characters and background on paper grids. For background it is wise to draw up a 40 x 25 square grid (make several blank copies if you can) and plan exactly what characters you'll place in every location. It's very much faster than 'thinking' on the keyboard.

11.2.3 Plan the action sequence

The most important (and most difficult) part of a game program like *Commando Sub* is the *action sequence.* It's the part where you usually have two or more objects moving in quick succession in response to controls, target hits or collisions.

It's also the part you should work out first. There's nothing worse than having your background ready, your instructions written, your crash sequences working and your score displays ready to go — with an action sequence that won't go right. You can feel very silly indeed!

The action sequence in any game consists of the ongoing movements, the user inputs and the various outcomes. Figure 11.1 shows the sequence diagramatically.

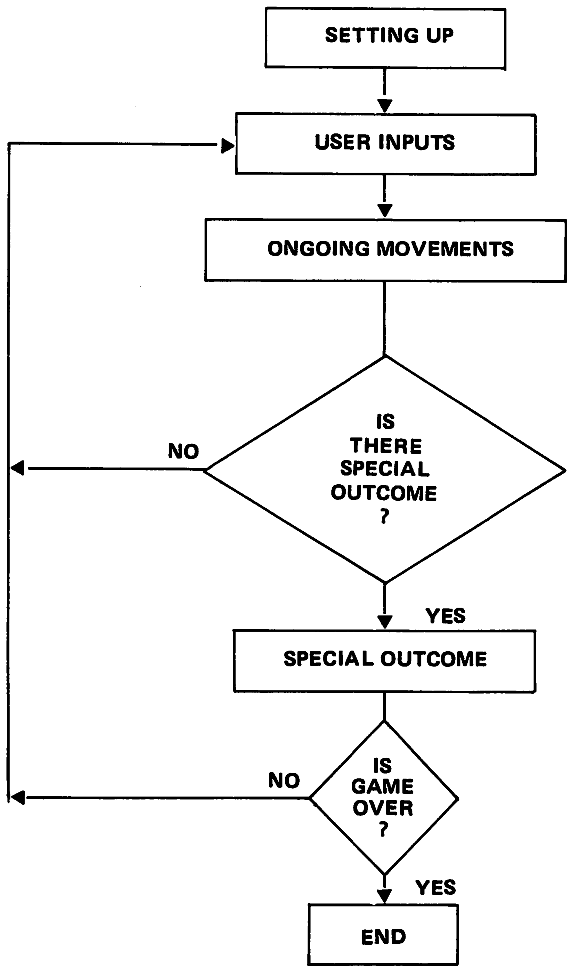


Figure 11.1 — the action sequence

This simple scenario may have to be duplicated two, three or more times to accommodate the number of moving objects. In *Commando Sub,* the ongoing movements are the sub in its path through the water, the missile in its fall to seabed, the destroyer in its continuous left and right travel, and the monster in its meandering about the entrance to the eastern gold. The inputs consist of the signals for ascending and diving, left and right. The signals may come either from the keyboard or a joystick. The outcomes are gold collection, collision, missile strike and successful return to home base.

To accommodate these moving parts and their various outcomes, we need to plan a *loop* or order of actions. I have chosen this sequence for *Commando Sub:*

- sub movement
- destroyer movement
- missile movement
- monster movement
- sub movement
- etc.

Destroyer and monster act randomly. We do not need to include collision checks in their movement sequences, since the sub movement sequence has its own collision check. Let us look more closely at the sub and missile movements in Figures 11.2 and 11.3.

Note: regular PRINT or POKE animation requires that you erase the character from its current position before you display it in its next position. With sprites on the C-64, however, the erasure is automatic. All you have to do is feed in the coordinates for the next position. This makes life a lot easier for the C-64 game programmer.

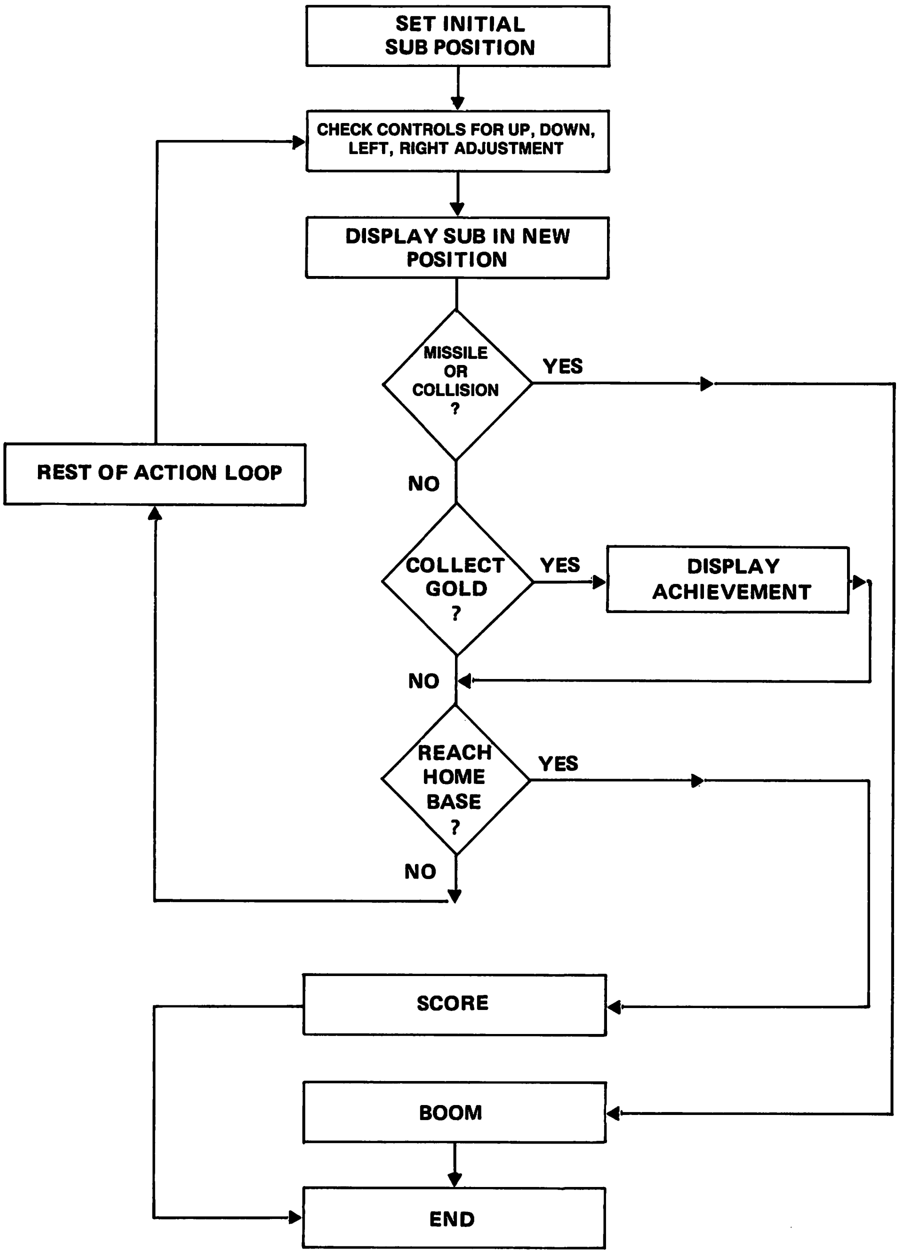


Figure 11.2 — sub movement sequence

11.2.4 Missile movement

Missile action is continuous in this game. The destroyer releases a new missile as soon as the previous one is spent.

Notice that we don't ask 'has the missile hit the sub?'. We need not bother with this check because we've already asked (in the submarine movement routine), 'has the sub hit the missile?' which is logically the same question.

Once the submarine sequence and the missile sequence are working, the other sprite movements fit in easily.

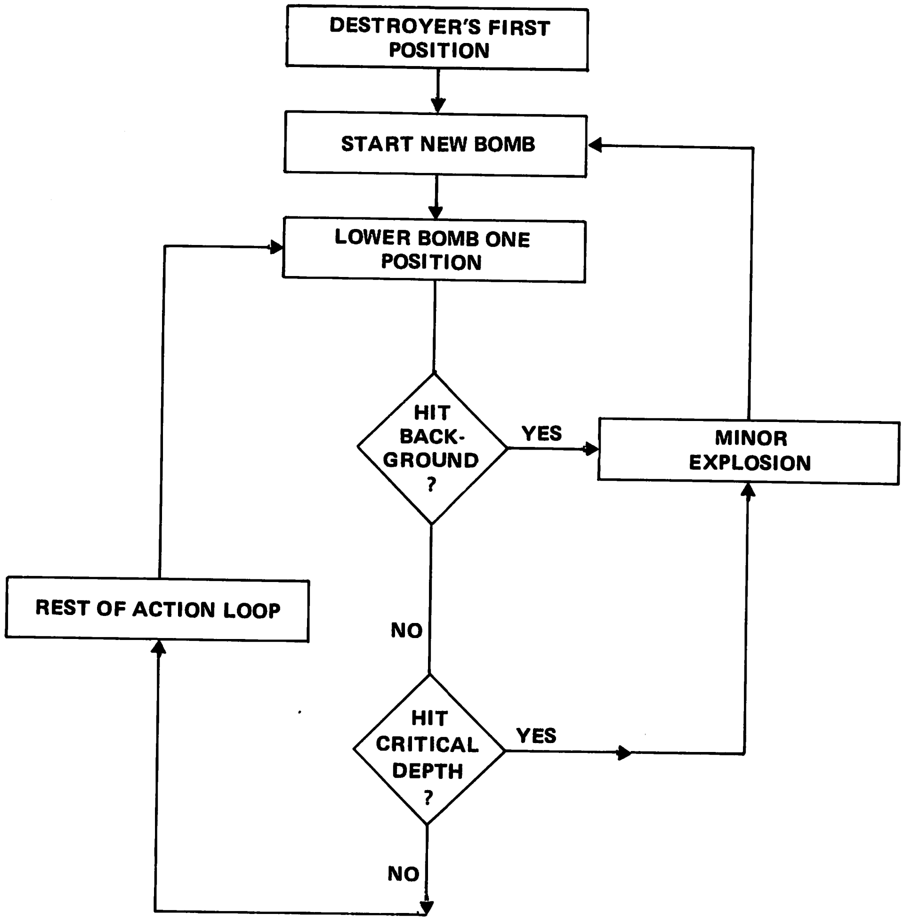


Figure 11.3 — missile movement sequence

Figure 11.4 will help you to draw up a plan for the game your own imagination has created.

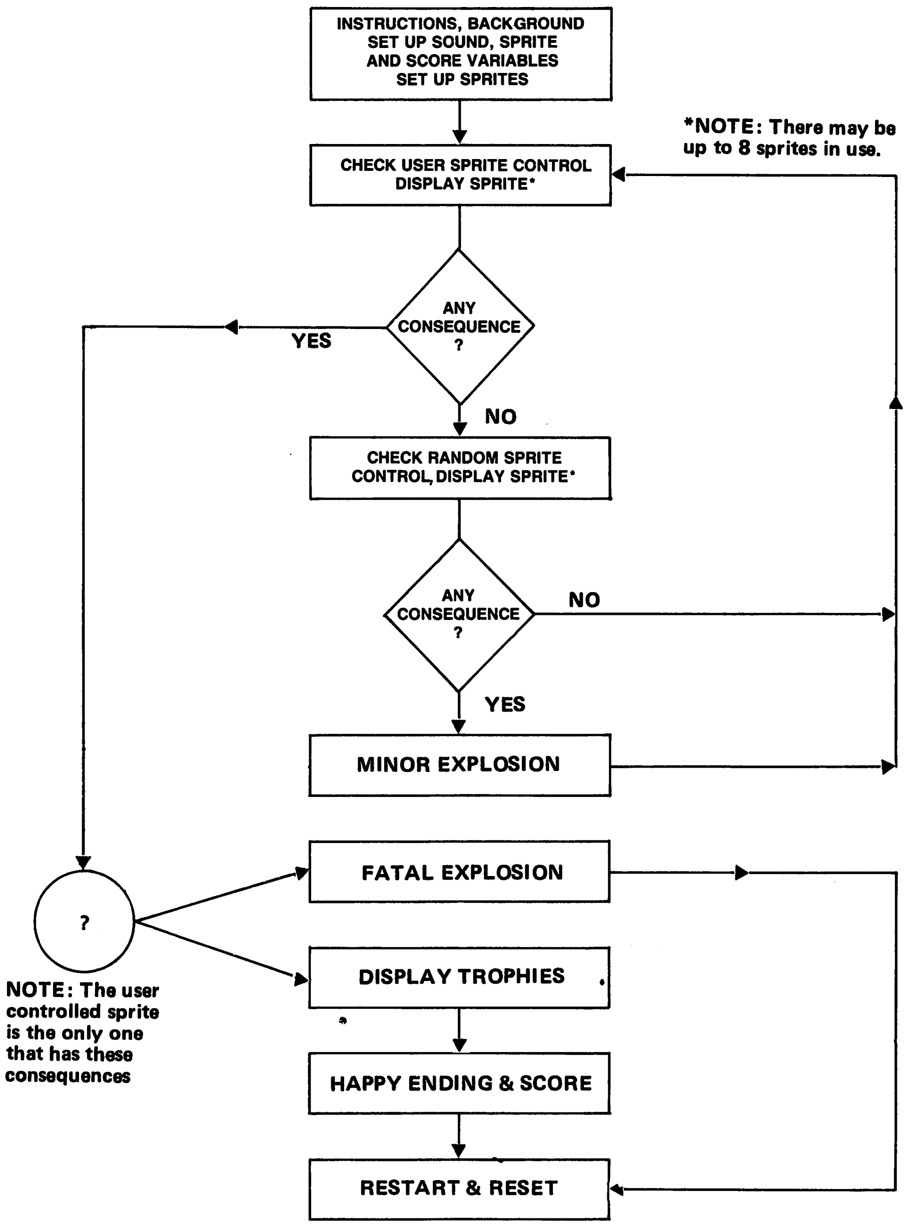


Figure 11.4 — the complete game plan

11.2.5 Write the code

Write your lines of code, *module by module.*

- SAVE your program (to tape or disk) as you complete each module and before you RUN it.
- RUN each module separately, as you complete it, to make sure that it works. Then RUN it in combination with any other modules you've already written. You may have to insert the word STOP at the end of a module to do this — so don't forget to remove the STOPs later!
- When you add a new variable, make sure its name is different from the names you've given to any other variables you've already used. Two variables are counted as being the same if their first two characters are the same. Test RUN to make sure.
- Avoid calling variables by names that include BASIC keywords — eg RUNWAY. Some others you should avoid are TI, TI$ and ST. They are reserved and have special meanings.
- Take special care with screen POKEs. Make sure that things can't run off the screen before you test RUN. If you do run off the screen, switch off the computer and reload your latest SAVE because important addresses are sure to have been fouled up by the run off.

11.2.6 Enhance the program

Now's the time to go back over the program, module by module, to insert the other things you've thought of to 'jazz up' your program — better graphics, extra hazards, player instructions and so on. As above, SAVE the program with its extra lines first and then RUN the whole program.

Appendix A
Commando Sub — the complete program

```
40 REM CLEAR HI SCORE STORAGE ADDRESS
45 POKE 49152,0
47 GOTO 4000:REM INSTRUCTIONS
50 REM BACKGROUND COLOR SCHEME
52 POKE53280,9:POKE53281,14:PRINT"♥";
56 FORF=1TO18
57 PRINT"▟                           ";:NEXT
58 GOSUB6000:REM GET ROCKS
60 REM FUEL SUPPLY
65 LT=2500
70 REM SOUND ABBREVIATIONS
72 VO=54296:PI=54273:DE=54277:GA=54276
73 REM VOLUME,PITCH,DECAY,GATE ADDRS VOICE 1
74 P2=54280:D2=54284:G2=54283
75 REM PITCH,DECAY & GATE ADDRS FOR VOICE 2
76 REM VOICE 2 USED IN SUB CRASH
78 POKE VO,5:REM VOLUME SETTING FOR SONAR
79 POKE GA,0:POKE G2,0:REM CLEAR SOUND GATES
80 REM SPRITE DEFINITIONS
90 SC=53287:REM SPRITE COLOR
91 BP=53275:REM BACKGROUND PRIORITY
92 EN=53269:REM SPRITE ENABLE - ON/OFF
93 SS=53264:REM SIDE OF SCREEN
94 FORF=0TO7:X(F)=53248+2*F:NEXT:REM X ADDRS
96 FORF=0TO7:Y(F)=53249+2*F:NEXT:REM Y ADDRS
```

```
100 REM SUB SPRITE
105 SUBMEM=15872:REM 1ST ADDR FOR SUB DATA
107 POKE 2040,248:
110 FOR F=0TO62
111 READ BY
112 POKE SUBMEM+F,BY
113 NEXT
120 DATA0,0,0,0,0,0,0,3,0,0,2,0,0,2,0
121 DATA0,15,0,63,143,0,6,143,0,3,15,7
122 DATA255,255,255,250,175,255,250,175
123 DATA255,127,255,252,0,0,0,0,0,0,0,0
124 DATA0,0,0,0,0,0,0,0,0,0,0,0,0,0
180 REM ASSIGN A COLOR
181 POKE SC+0,11:REM SPRITE COLOR 0
182 REM ASSIGN BACKGROUND PRIORITY
183 POKE BP,0:REM SUB IN FRONT OF BACKGROUND
184 REM ASSIGN INITIAL POSITION
185 POKE SS,0:REM LEFT OF SCREEN
186 POKE X(0),31:REM 31ST COLUMN
188 POKE Y(0),110:REM 110TH ROW
190 POKE EN,1:REM SET SPRITE 0 ON
192 U(0)=110:L(0)=230:REM UP/LOW LIMITS SUB
194 W(0)=31:E(0)=60:REM SUB EAST/WEST LIMITS
200 REM LEFT ISLAND
205 R1MEM=15936
207 POKE 2041,249:REM SPRITE POINTER #1
208 REM DIRECTED TO 249*64, THAT IS 15936
210 FOR F=0TO62
211 READ BY
212 POKE R1MEM+F,BY
213 NEXT
220 DATA3,0,0,3,240,0,7,248,0,15,252,0
221 DATA15,255,0,31,255,224,63,255,248
222 DATA63,255,255,127,255,255,127,255,252
223 DATA127,255,248,127,255,240
224 DATA255,255,224,255,255
225 DATA240,127,255,252,63,255,254
226 DATA0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
280 REM ASSIGN A COLOR
281 POKE SC+1,1:REM SPRITE COLOR 1
282 REM ICEBERG APPEARS IN FRONT OF SKY
283 REM BP STILL HAS ALL ZEROS FROM LINE 183
```

```
284 REM ASSIGN INITIAL POSITION
285 POKE SS,0:REM LEFT OF SCREEN
286 POKE X(1),56:REM 56TH COLUMN
288 POKE Y(1),114:REM 114TH ROW
290 POKE EN,3:REM SET SPRITE 0 & 1 ON
300 REM RIGHT ICEBERG
305 REM SPRITE DATA SAME AS FOR LEFT ICEBERG
306 REM SO SPRITE POINTER #2 POINTS
307 REM TO SAME ADDRESS AS SPRITE POINTER #1
308 POKE 2042,249:REM SPRITE POINTER #2
309 REM DIRECTED TO 249*64, THAT IS 15936
380 REM ASSIGN A COLOR
381 POKE SC+2,1:REM SPRITE COLOR 2
382 REM ICEBERG APPEARS IN FRONT OF SKY
383 REM BP STILL HAS ALL ZEROS FROM LINE 183
384 REM ASSIGN INITIAL POSITION
385 POKE SS,4:REM SPRITE 2 RIGHT OF SCREEN
386 POKE X(2),16:REM 256+16=272ND COLUMN
387 REM OR 16TH COLUMN OF RIGHT SIDE
388 POKE Y(2),114:REM 114TH ROW
390 POKE EN,7:REM SET SPRITE 0,1 & 2 ON
400 REM CENTRAL REEF
405 REM SPRITE DATA SAME AS FOR LEFT ICEBERG
406 REM SO SPRITE POINTER #3 POINTS
407 REM TO SAME ADDRESS AS SPRITE POINTER #1
408 POKE 2043,249:REM SPRITE POINTER #3
409 REM DIRECTED TO 249*64, THAT IS 15936
480 REM ASSIGN A COLOR
481 POKE SC+3,9:REM SPRITE COLOR 3
482 REM ICEBERG APPEARS IN FRONT OF SKY
483 REM BP STILL HAS ALL ZEROS FROM LINE 183
484 REM SPRITE 3 ALREADY LEFT OF SCREEN
485 REM SS HAS 0 IN BIT 3 FROM LINE 385
486 POKE X(3),165:REM 165TH COLUMN
488 POKE Y(3),124:REM 124TH ROW
490 POKE EN,15:REM SET SPRITE-0,1,2 & 3 ON
500 REM DESTROYER
505 DSMEM=16000:REM 1ST ADDRESS FOR DATA
507 POKE 2044,250:REM SPRITE POINTER #4
508 REM DIRECTED TO 250*64, THAT IS 16000
510 FOR F=0TO62
511 READ BY
```

```
512 POKE DSMEM+F,BY
513 NEXT
520 DATA0,6,0,0,6,0,3,198,0,1,134,0,1,134,0
521 DATA15,255,240,6,102,96,246,102,111
522 DATA55,255,236,255,255,255,247,189,239
523 DATA99,24,198,119,189,238,63,255,252
524 DATA0,0,0,0,0,0,0,0,0,0,0,0
525 DATA0,0,0,0,0,0,0,0,0
580 REM ASSIGN A COLOR
581 POKE SC+4,6:REM SPRITE COLOR 4
582 REM DESTROYER APPEARS IN FRONT OF SKY
583 REM BP STILL HAS ALL ZEROS FROM LINE 183
584 REM SPRITE 4 ALREADY LEFT OF SCREEN
585 REM SS HAS 0 IN BIT 4 FROM LINE 385
586 POKE X(4),204:REM 204TH COLUMN
588 POKE Y(4),108:REM 108TH ROW
590 POKE EN,31:REM SET SPRITE 0,1,2,3 & 4 ON
594 W(4)=48:E(4)=47:REM EAST & WEST LIMITS
600 REM MONSTER
605 MOMEM=16064
607 POKE 2045,251:REM SPRITE POINTER #5
608 REM DIRECTED TO 251*64, THAT IS 16064
610 FOR F=0TO62
611 READ BY
612 POKE MOMEM+F,BY
613 NEXT
620 DATA0,0,0,0,31,192,3,63,224,6,102,97
621 DATA28,102,97,56,127,227,112,127,227
622 DATA112,127,195,224,255,199,224,255,207
623 DATA224,255,254,225,239,254,227,238,252
624 DATA255,206,121,255,158,3,255,30,14,126
625 DATA30,30,124,30,60,0,31,248,0,15,240
626 DATA0,7,224
680 REM ASSIGN A COLOR
681 POKE SC+5,13:REM SPRITE COLOR 5
682 POKEBP,32:REM ONLY BIT 5 = 1
683 REM SO ONLY SPRITE 5 GOES BEHIND BKGRND
684 POKESS,36:REM SPRITES 2,5 RIGHT OF SCR
685 REM SO BITS 2&5 TOTAL 36: 4+32=36
686 POKE X(5),0:REM IST COLUMN ON RIGHT SIDE
688 POKE Y(5),242:REM 242ND ROW
690 POKE EN,63:REM SET SPRITES 0 TO 5 ON
```

```
692 U(5)=187:L(5)=242:REM UP & LOW LIMITS
694 W(5)=0:E(5)=45:REM EAST & WEST LIMITS
700 REM MISSILE
705 MISMEM=16128
707 POKE 2046,252:REM SPRITE POINTER #6
708 REM DIRECTED TO 252*64, THAT IS 16128
710 FOR F=0TO62
711 READ BY
712 POKE MISMEM+F,BY
713 NEXT
720 DATA0,8,0,0,8,0,0,8,0,0,8,0
721 DATA0,0,0,0,0,0,0,0,0,0,0,0,0,0
722 DATA0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
723 DATA0,0,0,0,0,0,8,0,0,8,0,0,8,0,0,8,0
780 REM ASSIGN A COLOR
781 POKE SC+6,0:REM SPRITE COLOR 6
782 POKEBP,96:REM ONLY BITS 5 & 6 = 1
783 REM SPRITES 5 & 6 GO BEHIND BACKGROUND
784 REM SPRITE 6 LEFT OF SCREEN
785 REM SS HAS 0 IN BIT 6 FROM LINE 385
786 POKE X(6),204:REM DESTROYERS INIT COLUMN
788 POKE Y(6),123:REM 16 ROWS UNDER DEST ROW
790 POKE EN,127:REM SET SPRITES 0-6 ON
792 L(6)=206:REM JUST ABOVE SUB ON MUD
794 REM E & W LMTS CONTROLLED BY DEST LMTS
800 REM EXPLOSION
805 EXMEM=16192
810 FOR F=0TO62
811 READ BY
812 POKE EXMEM+F,BY
813 NEXT
820 DATA0,0,0,0,12,3,0,12,3,128,12,12,224,12
821 DATA56,56,8,224,44,8,128,11,190,192
822 DATA2,215,0,0,151,192,10,213,240,233,87
823 DATA192,3,87,0,3,214,192,2,165,128
824 DATA14,47,224,10,32,56,44,32,14,48,32
825 DATA131,192,32,192,192,48,48
880 REM ASSIGN A COLOR
882 POKE 53285,1:REM MULTI COLOR 0
883 POKE 53286,7:REM MULTI COLOR 1
884 REM 3RD COLOR = SPRITE 6 COLOR (BLACK)
885 REM WILL BE MADE RED DURING EXPLOSION
```

```
1000 REM CONTROLLING SUB
1005 F=PEEK(53278):F=PEEK(53279)
1006 REM ZERO COLLISION CHK (1ST TIME ONLY)
1010 KE=PEEK(197):REM SCAN KEYBOARD
1012 LT=LT-5:IF LT=0 THEN 3000
1014 PRINT"▒▓FUEL:▒"LT"▒ "
1015 IF KE=6 THEN GOSUB2020:REM F5 UP
1020 IF KE=3 THEN GOSUB2040:REM F7 DOWN
1025 IF KE=10 THEN GOSUB2060:REM A WEST
1030 IF KE=18 THEN GOSUB2080:REM D EAST
1040 GOSUB2100:REM COLLISION CHECK
1045 IF CO THEN 2800:REM CRASH
1046 IF PY=126 AND PX=60 THEN GOSUB 1410
1048 IF PY=214 AND PX=166 THEN GOSUB1420
1050 GOSUB2252:REM DESTROYER MOVE
1300 REM MONSTER MOVEMENT
1310 MM=INT(RND(3)*5)+1
1320 ON MM GOSUB2422,2442,2462,2482,1340
1330 GOTO2542:REM GOTO MISSILE MOVE
1340 RETURN
1400 REM GETTING THE GOLD
1410 IF(SP AND 1)=0 THEN RETURN
1411 REM LINE 1410 CHECKS SIDE OF SCREEN
1412 POKE 1463,32:REM SPACE OVER GOLD
1414 PRINT"▒",,,"▒▓EAST GOLD";:TG=1:RETURN
1420 IF(SP AND 1) THEN RETURN
1421 REM LINE 1420 CHECKS SIDE OF SCREEN
1422 POKE1884,32:REM SPACE OVER GOLD
1424 PRINT"▒",,"▒▓▓▓DEEP GOLD":BG=1:RETURN
2000 REM SUB MOVEMENT (= SPRITE NUMBER 0)
2020 REM SPRITE UP
2022 PY=PEEK(Y(0)):REM PRESENT Y POSN
2024 IF PY=U(0) THEN RETURN:REM UPPER LIMIT
2025 POKE Y(0),PY-4:RETURN:REM RAISE Y POSN
2040 REM SPRITE DOWN
2042 PY=PEEK(Y(0)):REM PRESENT Y POSN
2044 IF PY=L(0) THEN RETURN:REM LOWER LIMIT
2045 POKE Y(0),PY+4:RETURN:REM LOWER
2060 REM SPRITE WEST
2062 PX=PEEK(X(0)):REM PRESENT X POSN
2064 SP=PEEK(SS):PS=SPAND1:REM SIDE OF SCR
2066 IF PX<=W(0)ANDPS=0 THEN2910:REM W LIMIT
```

```
2068 IF PX=0 THEN 2073
2069 REM LINE 2068 IS FOR CROSSING TO WEST
2070 POKE X(0),PX-5:RETURN:REM GO WEST
2073 POKE X(0),251:POKE SS,PEEK(SS)AND254
2075 F=PEEK(53278):F=PEEK(53279):RETURN
2076 POKE X(0),0:POKESS,PEEK(SS)OR1:GOTO2075
2080 REM SPRITE EAST
2082 PX=PEEK(X(0)):REM PRESENT X POSN
2084 SP=PEEK(SS):PS=SPAND1:REM SIDE OF SCR
2086 IF PX=E(0)ANDPS THENRETURN:REM EAST LMT
2088 IF PX=251 THEN 2076:REM CROSS TO EAST
2090 POKE X(0),PX+5:RETURN:REM GO EAST
2100 REM SUB COLLISION CHECK
2110 REM COLLIDE WITH BACKGROUND?
2112 PX=PEEK(X(0))
2115 PY=PEEK(Y(0)):IF PY<123 THEN 2140
2120 IF PEEK(53279)AND1 THEN CO=1
2130 REM COLLIDE WITH OTHER SPRITE?
2140 IF PEEK(53278)AND1 THEN CO=1
2145 F=PEEK(53279)
2150 RETURN
2200 REM DESTROYER MOVEMENT
2250 REM SELECT DIRECTION
2252 PX=PEEK(X(4)):REM PRESENT X POSN
2254 SP=PEEK(SS):PS=SPAND16:REM SIDE OF SCR
2255 IF PEEK(56324)>240 THEN SE=1-SE
2259 IF SE THEN 2286:REM SE=1  SHIP GO EAST
2260 REM SPRITE WEST
2266 IF PX<W(4)ANDPS=0 THENSE=1:REM WEST LMT
2268 IF PX=0 THEN POKE X(4),252:GOTO2272
2269 REM LINE 2268 IS FOR CROSS TO WEST SIDE
2270 POKE X(4),PX-4:RETURN:REM GO WEST
2272 POKESS,SPAND239:GOTO2075:REM CHNGE SIDE
2280 REM SPRITE EAST
2286 IF PX>E(4)ANDPS THEN SE=0:REM EAST LMT
2288 IF PX=252 THEN POKE X(4),0:GOTO 2292
2289 REM LINE 2288 IS FOR CROSS TO WEST SIDE
2290 POKE X(4),PX+4:RETURN:REM GO EAST
2292 POKE SS,SPOR16:GOTO2075:REM CHANGE SIDE
2400 REM MONSTER MOVEMENT (= SPRITE NO. 5)
2420 REM SPRITE UP
2422 PY=PEEK(Y(5)):REM PRESENT Y POSN
```

```
2424 IF PY<U(5) THEN RETURN:REM UPPER LIMIT
2425 POKE Y(5),PY-3:RETURN:REM RAISE Y POSN
2440 REM SPRITE DOWN
2442 PY=PEEK(Y(5)):REM PRESENT Y POSN
2444 IF PY>L(5) THEN RETURN:REM LOWER LIMIT
2445 POKE Y(5),PY+1:RETURN:REM LOWER
2460 REM SPRITE WEST
2462 PX=PEEK(X(5)):REM PRESENT X POSN
2466 IF PX=W(5) THEN RETURN:REM WEST LIMIT
2470 POKE X(5),PX-3:RETURN:REM GO WEST
2480 REM SPRITE EAST
2482 PX=PEEK(X(5)):REM PRESENT X POSN
2486 IF PX=E(5) THEN RETURN:REM EAST LIMIT
2490 POKE X(5),PX+3:RETURN:REM GO EAST
2500 REM MISSILE MOVEMENT (= SPRITE NO. 6)
2540 REM SPRITE DOWN
2542 PY=PEEK(Y(6)):REM PRESENT Y POSN
2544 IF PY>L(6) THEN 2700:REM LOWER LIMIT
2545 POKE Y(6),PY+6:REM LOWER
2560 REM SONAR PING
2562 POKE DE,9
2564 POKE PI,200:POKE GA,33
2565 REM SET HI PI & WAVEFORM 32 (+1 FOR ON)
2566 FOR F=1TO20:NEXT
2568 POKE GA,0:REM CEASE SOUND
2600 REM MISSILE COLLISION CHECK
2610 REM COLLIDE WITH BACKGROUND?
2620 CM=0:IF PEEK(53279)AND64 THEN CM=1
2650 IF CM=0 THEN 1010:REM BACK TO SUB MOVE
2700 REM MISSILE EXPLODE
2705 POKEVO,10:POKEDE,11:POKEPI,3:POKEGA,129
2706 REM LOW PITCH & NOISE GENERATOR
2710 POKE 53281,4:REM SCREEN COLOR
2715 POKE 53276,64:REM SPR 6 MULTI COL MODE
2717 POKE SC+6,2:REM SPRITE COLOR 6
2720 POKE 2046,253:REM MISSILE POINTER
2725 FOR F=1TO100:NEXT
2726 POKE EN,63:REM DISABLE SPRITE 6
2727 REM ENABLE ONLY SPRITES 0-5
2730 POKE 53281,14:POKE53280,9:REM SCR COL
2735 POKE SC+6,0:REM SPRITE COLOR 6
2740 POKE 2046,252:REM MISSILE PTR TO NORMAL
```

```
2745 POKE 53276,0:REM ALL SPRS TO HIRES MODE
2750 POKE X(6),PEEK(X(4)):REM DESTR COLUMN
2755 PS=PEEK(SS):POKE SS,PSAND191
2756 IF PSAND16 THEN POKE SS,PSOR64
2760 POKE Y(6),123:REM 16 ROWS BELOW DESTR
2765 POKE EN,127:REM RE-ENABLE SPRITES 0-6
2770 POKE GA,0:REM KILL SOUND
2780 GOTO 1010:REM BACK TO SUB MOVE
2800 REM SUB EXPLODE
2805 POKEDE,14:POKEVO,15:POKEPI,3:POKEGA,129
2806 REM LOW PITCH & NOISE GENERATOR
2807 POKED2,14:POKEP2,3:POKEG2,129
2808 REM LO PI & NOISE GENERATOR ON VOICE 2
2810 POKE 53281,4:POKE53280,0:REM SCR COLOR
2815 POKE 53276,1:REM SPR 0 - MUL'COLOR MODE
2817 POKE SC+0,2:REM SPRITE COLOR 0
2820 POKE 2040,253:REM SUB POINTER
2821 REM TEMP REDIRECTED TO BOOM SPRITE
2825 FOR F=1TO200:NEXT
2827 POKE 53277,1:REM POKE SPRITE WIDENER
2828 REM WITH BIT 0 ON (FOR SPRITE 0)
2830 FOR F=1TO200:NEXT
2835 POKE 53271,1:REM POKE SPRITE DEEPENER
2836 REM WITH BIT 0 ON (FOR SPRITE 0)
2837 FOR F=1TO200:NEXT
2845 POKE 53281,14:POKE53280,9
2846 REM SCREEN COLOR BACK TO NORMAL
2850 PRINT"▒▒▒▒▒▒▒▒▒▒▒▒▒▒SUB DESTROYED"
2860 GOTO 3052
2900 REM WEST LIMIT FOR SUB
2910 IF PX<W(0)THEN POKE Y(0),110:REM UP LMT
2920 PY=PEEK(Y(0)):REM PRESENT Y POSN
2924 IF PY<>U(0) THEN RETURN:REM UPPER LIMIT
2930 IF PX=6 THEN POKE X(0),1:GOTO 3000
2940 POKE X(0),PX-5:RETURN:REM GO WEST
3000 REM FINISH
3002 IF LT THEN 3010
3005 PRINT"▒▒▒▒▒▒▒▒▒▒▒▒▒OUT OF FUEL"
3006 PRINT"▒▒RADIO FOR HELP":GOTO 3052
3010 PRINT"▒▒▒▒▒▒▒▒▒▒▒▒▒RETURNED TO BASE"
3020 PT=800*BG+2000*TG
3025 IF PT=2800 THEN PT=4000
```

```
3027 IF PT=0 THEN PRINT"▓▓EMPTY HANDED"
3028 IF PT=0 THEN PRINT"▓▓NO SCORE":GOTO3052
3030 IF BG THEN PRINT"▓▓WITH GOLD FROM DEEP"
3035 IF TG THEN PRINT"▓▓WITH GOLD FROM EAST"
3040 PRINT"▓▓FUEL SAVED ="LT"LITRES"
3045 PT=PT+LT:PT=20*INT(PT/20)
3050 PRINT"▓▓YOU SCORED"PT"POINTS"
3052 HI=PEEK(49152)*20
3055 PRINT"▓▓PREVIOUS RECORD ="HI
3060 IF PT>HI THEN POKE 49152,PT/20
3070 PRINT"▓▓PRESS BAR FOR MORE"
3075 GET A$:IF A$<>"" THEN 3075
3078 GET A$:IF A$<>" " THEN 3078
3079 POKE GA,0:POKE G2,0:POKE 53277,0
3080 REM KILL SOUND
3081 POKE53277,0:POKE 53271,0
3082 REM KILL WIDEN & DEEPEN
3083 POKE 53269,0:REM DISABLE ALL SPRITES
3084 POKE 53276,0:REM ALL SPRS TO HIRES MODE
3085 RUN 47:REM RESTART
4000 REM INSTRUCTIONS
4010 POKE 53280,0:POKE 53281,0:PRINT"▓";
4015 PRINT"▓          **** SUB COMMANDO ****"
4020 PRINT"▓▓YOU ARE AT WAR WITH";
4021 PRINT" THE IMPERIAL FORCES.";
4022 PRINT"YOUR GOLD IS HIDDEN UNDER ";
4023 PRINT"THE SEA.":PRINT"IT IS NEEDED ";
4024 PRINT"TO CONTINUE THE WAR EFFORT.";
4025 PRINT"▓▓THERE ARE TWO CACHES OF GOLD -"
4026 PRINT"ONE IN THE DEEP ";
4027 PRINT"AND ONE IN THE EAST."
4030 PRINT"▓▓AS A COMMANDO ";
4031 PRINT"YOU ARE EXPECTED TO"
4035 PRINT"RISK ALL TO RETRIEVE THE GOLD."
4040 PRINT"▓▓A DESTROYER PATROLS THE AREA.";
4041 PRINT" ITS DEPTH CHARGES EXPLODE ";
4042 PRINT"BEFORE REACHING BOTTOM."
4050 PRINT"▓▓AVOID THE SEA ";
4051 PRINT"MONSTER THAT GUARDS THE"
4055 PRINT"ENTRANCE TO THE EASTERN GOLD.▓"
4060 PRINT"▓▓YOUR ORDERS ARE TO RETURN WITH";
4061 PRINT" THE GOLD,SURFACE AT WEST BASE";
```

```
4064 PRINT" (GREY) AND ENTER";
4065 PRINT"   THE BASE (GO OFF SCREEN)"
4066 PRINT"▥__________________________________________";
4067 PRINT"______________";
4068 PRINT"▨ CONTROLS: A-LEFT";
4069 PRINT" D-RIGHT F5-UP F7-DOWN "
4070 PRINT"▙▐▐▐▐▐▐▐▐▐▐▐▐PRESS BAR TO START▜";
4080 GET A$:IF A$<>" " THEN 4080
4085 RUN 50:REM START
```

```
6000 REM BACKGROUND DETAIL
6005 PRINT"[illegible] "
6010 PRINT"[illegible]";
6012 PRINT"[illegible]";
6014 PRINT"[illegible]"
6016 PRINT"[illegible]"
6018 PRINT"[illegible]"
6020 PRINT"[illegible]"
6022 PRINT"[illegible]"
6024 PRINT"[illegible]"
6026 PRINT"[illegible] ";
6028 PRINT"[illegible] ";
6030 PRINT"[illegible] ";
6032 PRINT"[illegible]  ";
6034 PRINT"[illegible]  ";
6036 PRINT"       [illegible]     ";
6038 PRINT"        [illegible]     ";
6040 FORF=1984TO2023:POKEF,160:POKEF+54272,9:NEXT
6042 PRINT"[illegible]";:RETURN
```

Appendix B
Joystick control

If you have a joystick and would rather use it than the function keys to control the game, you can change the control procedure from PEEKing the keyboard to PEEKing the movement of the joystick.

Plug your joystick into the *second port* on the right hand side of your computer (it's marked CONTROL PORT 2) and type the following line. It's not part of the program itself for *Commando Sub* — we're doing it just to demonstrate something about the joystick.

```
950 JS=PEEK(56320):PRINT JS:GOTO 950
```

Line 950 sets up a variable name JS and gives it the value of the contents of address 56320. 56320 holds the joystick 'POKEs' for its various positions. The rest of the line tells the computer to PRINT on the screen the values it finds for JS — and to keep doing it till we tell it to STOP.

So when you type RUN 950 you'll see a continuous stream of numbers on the screen. As you watch these numbers, move the joystick into its various positions. The joystick positions we're interested in are up, down, left and right. For these positions we get the following numbers:

● UP	125
● DOWN	126
● LEFT	123
● RIGHT	119

Now we can fold these numbers straight into our program, substituting them for the values we have given our KEy variable:

```
1010 JS = PEEK (56320)
1015 IF JS = 125 THEN GOSUB 2020
1020 IF JS = 126 THEN GOSUB 2040
1025 IF JS = 123 THEN GOSUB 2060
1030 IF JS = 119 THEN GOSUB 2080
```

Note: pressing the firing button on the joystick will alter these values and therefore serve as a *brake* on the sub's movement.

Appendix C
The binary to decimal converter

Here is a short program you might like to use to convert binary numbers to their decimal equivalents. If you SAVE it after you've keyed it in, it will be available whenever you need it. Just remember to NEW it out before you start to write the program into which you're going to put the decimal numbers it has calculated for you.

When you RUN the program, the computer will prompt you with a question mark (?). All you do is enter your 8 bit binary number and press the RETURN key. The computer converts it to its decimal equivalent and then prompts with another question mark. When you've finished, you stop the program with RUN/STOP and RESTORE.

```
10 PRINT,"CONVERTER"
15 PRINT,"————————————▒"
20 PRINT,"8-BIT                   EQUIVALENT"
22 PRINT,"BINARY                   DECIMAL"
24 PRINT,"NUMBER                   NUMBER▒▒"
30 INPUT"▌▌▌▌▌▌▌▌▌";BN$
40 DN=0:REM DECIMAL NUMBER
42 FORF=0TO7:REM POWERS OF 2
44 N=8-F:REM POSN IN BN$ STRG
50 A$=MID$(BN$,N,1):REM NTH CHARACTER
55 A=VAL(A$):REM NUMERICAL VALUE
60 DE=A*2↑F:REM DECIMAL EQUIVALENT
65 DN=DN+DE:REM ACCUMULATE
70 NEXT
75 PRINT,,,"⌐"STR$(DN)"▒▒":REM DISPLAY DECIMAL
80 GOTO30:REM GET NEXT BINARY NUMBER
```

Appendix D
The Super Sprite Maker

The *Super Sprite Maker* is a program that lets you design your sprites (on a large scale grid) on the screen of your TV monitor. When you've finished your design, the program converts it to a set of 63 decimal numbers — the 63 bytes you need for each sprite.

After LOADing the program (and keying RUN) all you do is follow the instructions on the screen. There's not much more to it than inserting blobs wherever you fancy (locating the spot with the cursor keys and then pressing any letter key) and pressing RETURN when you've finished. If you want to erase a blob (before you've pressed RETURN) you locate the spot with the cursor and press the space bar.

The program is written in such a way that you can either print out the sprite DATA (if you have a printer) or else SAVE it on cassette or disk until you need it. When you want to use the DATA in another program (if you've SAVEd it), all you do is LOAD the DATA lines you've generated and then add the rest of the program (with higher line numbers). It can save you a lot of calculating and typing!

If you've already written your program and want to use the *Super Sprite Maker* to design or redesign sprites for it, simply type the DATA into your program from the hardcopy printout or from a hand written copy of the screen display.

```
80 REM   *** SUPER SPRITE MAKER ***
81 REM
82 REM  ******  INSTRUCTIONS  *******
83 REM
84 REM AFTER YOU HAVE MADE A SPRITE,
85 REM YOU CAN PRINT OUT THE DATA
86 REM FOR FUTURE USE, OR ELSE YOU CAN
87 REM STORE IT IN THIS PROGRAM.
88 REM AFTER STORING EACH ONE, SIMPLY
89 REM RUN THIS PROGRAM AGAIN.
90 REM WHEN YOU HAVE MADE ALL YOUR
91 REM SPRITES, DELETE LINES 80 TO 790
92 REM THEN SAVE WHATS LEFT - NAMELY THE
93 REM DATA LINES 0 TO 76 (OR 66 OR ETC).
94 REM THEN LOAD AND PROCEED TO BUILD YOUR
95 REM PROGRAM ON TOP,STARTING AT LINE 100.
96 REM YOU MUST CALL UP YOUR SPRITES
97 REM IN  THE SAME ORDER AS YOUR DATA.
100 POKE53280,12:POKE53281,12
105 JU=54272:POKE650,128
110 R$="▶▶▶▶▶▶▶▶▶▶▶▶▶▶▶▶▶▶▶▶▶▶▶▌"
120 PRINT"▣▶▶▶▣SUPER SPRITE MAKER"
125 PRINT"▶▶▶▣──────────────────────"
130 PRINT"▣▣"R$"USE CURSORS"
135 PRINT R$"▣USE ANY "
140 PRINT R$"LETTER TO":PRINT R$"DRAW"
150 PRINT R$"▣USE SPACE BAR"
155 PRINT R$"TO ERASE"
160 PRINT R$"▣HIT RETURN"
165 PRINT R$"WHEN FINISHED
170 PRINT"▣▣▣";
180 FORF=1TO21
190 PRINT"■.........................":NEXT
200 LO=1104
210 POKELO+JU,0:FORF=1TO10:NEXT
220 GETA$:IFA$=CHR$(17)THENGOSUB310
230 IFA$=CHR$(29)THENGOSUB330
240 IFA$=CHR$(145)THENGOSUB350
250 IFA$=CHR$(157)THENGOSUB370
260 IFA$>"@"ANDA$<"["THEN POKELO,81:GOSUB330
270 IFA$=" "THEN POKELO,46:GOSUB330
280 IFA$=CHR$(13)THEN 390
```

```
290 POKE LO+JU,12:FORF=1TO50:NEXT
300 A$="":GOTO210
310 IFLO>1903THEN RETURN
320 LO=LO+40:RETURN
330 IF(LO-47)/40=INT((LO-47)/40)THEN RETURN
340 LO=LO+1:RETURN
350 IFLO<1144THEN RETURN
360 LO=LO-40:RETURN
370 IF(LO-24)/40=INT((LO-24)/40)THEN RETURN
380 LO=LO-1:RETURN
390 REM
400 PRINT"▒"R$"▒▒▒▒▒▒▒▒▒▒▒▒▒STORING"
405 PRINT R$"PICTURE"
410 REM STORING DATA
420 MM=30000
430 FORF=1103 TO 1903 STEP40
440 FORG=0 TO 16 STEP8
450 TL=0:UT=F+G:FORH=7TO0 STEP-1
460 UT=UT+1
470 IF(PEEK(UT)=46)THEN 490
480 TL=TL+2↑H
490 NEXT:POKE(MM+F+G),TL
500 NEXT:NEXT
510 PRINT"▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒"R$"▒PRINT OUT (P)"
520 PRINT R$"SHOW DATA (D)"
525 PRINT R$"NEW SPRITE (N)████";
530 FORF=1TO50:NEXT:PRINT"▒(N)████";
535 FORF=1TO50:NEXT
540 GETA$:IFA$="P"ORA$="D"ORA$="N"THEN 550
545 PRINT"█(N)████";:GOTO530
550 IFA$<>"D"THEN700
560 PRINT:PRINT R$"▒TYPE SPRITE"
565 PRINT R$"NUMBER:";
570 GETB$:IFB$<"0"ORB$>"7"THEN 570
580 SP=VAL(B$)
590 PRINT"▒▒▒▒"
600 FORI=1103TO1823STEP120
610 LN=SP*10+(I-1103)/120:PRINT"▒██"LN;"▒▒♣";
620 FORF=I TO I+80 STEP40
630 FORG=0 TO 16 STEP8
640 PRINT MID$(STR$(PEEK(F+G+MM)),2)",";
650 NEXT:NEXT:PRINT"█▌ ▒":NEXT
```

```
660 PRINT"▨▶▶▶▶▶▶▶TO PRESERVE DATA, ";
665 PRINT"HIT ▨RETURN▨"
670 PRINT"▨KEY WITH CURSOR ON EACH ";
675 PRINT"▨RED▨ NUMBER▨▨";
680 END
690 GOTO690
700 IFA$<>"P"THEN 790
710 OPEN4,4
720 PRINT#4,"SPRITE DATA"
730 PRINT#4,"━━━━━━━━━━━━━━━━━━━━━━"
740 FORF=1103 TO 1903 STEP40
750 FORG=0 TO 16 STEP8
760 PRINT#4,PEEK(F+G+MM);
770 NEXT:NEXT
780 PRINT#4:CLOSE4:GOTO510
790 GOTO100
```

Appendix E
The symbols in this book

Here's what they mean and how to get them:

"▚▚▚"	CURSOR DOWN	VERTICAL CRSR
"▐▐▐"	CURSOR RIGHT	HORIZONTAL CRSR
"⌶⌶⌶"	CURSOR UP	SHIFT & VERT. CRSR
"▮▮▮"	CURSOR LEFT	SHIFT & HORIZ. CRSR
"▞▞▞"	TOP OF SCREEN	CLR/HOME
"⌶⌶⌶"	CLEAR SCREEN	SHIFT & CLR/HOME
"■■■"	BLACK	CTRL & 1
"▜▜▜"	WHITE	CTRL & 2
"▙▙▙"	RED	CTRL & 3
"▲▲▲"	CYAN	CTRL & 4
"▩▩▩"	PURPLE	CTRL & 5
"▮▮▮"	GREEN	CTRL & 6
"▬▬▬"	BLUE	CTRL & 7

"■■■"	YELLOW	CTRL & 8
"■■■"	ORANGE	C= & 1
"■■■"	BROWN	C= & 2
"■■■"	PINK	C= & 3
"■■■"	DARK GREY	C= & 4
"■■■"	MID GREY	C= & 5
"■■■"	LIGHT GREEN	C= & 6
"■■■"	LIGHT BLUE	C= & 7
"■■■"	LIGHT GREY	C= & 8

Many symbols (eg &, $, and ?) appear at the top of the key and require SHIFT.

The fun way to learn about *sprites* — the wonderful programmable object blocks which are a feature of the Commodore 64 computer — is to write a stimulating game that demonstrates how everything works.

How to write exciting games for the Commodore 64 focuses on the development of just one game, *Commando Sub*, and explains in detail every step needed to create its graphics, sound and action.

You are put at the helm of a mini submarine. Your mission is to recover two caches of gold from under the sea and keep it safe from the reach of the enemy. Seven sprites are featured — moving and stationary, high resolution and multicolor, constant and expanding — and all the sound effects needed to get the adrenalin really flowing.

Some games books give you listings for lots of programs and explain them in language only the experienced programmer can understand. This book is written in easy-to-understand language and by the time you've worked your way through it you'll know enough about programming in BASIC on the Commodore 64 to be able to write your own games using sprite graphics.

Pitman
ISBN 0 85896 181 4